# A Methodology for Secure Interactive Systems

Gerd Beuster

Gerd Beuster

# A Methodology for Secure Interactive Systems

Vom Promotionsausschuss des Fachbereichs 4: Informatik der Universität Koblenz-Landau zur Verleihung des akademischen Grades Doktor der Naturwissenschaften (Dr. rer. nat.) genehmigte Dissertation. Die wissenschaftliche Aussprache fand am 22. April 2008 statt.

| | |
|---|---|
| Vorsitzender des Promotionsausschusses: | Prof. Dr. Dieter Zöbel |
| Vorsitzender der Promotionskommission: | Prof. Dr. Gianfranco Walsh |
| Berichterstatter: | Prof. Dr. Bernhard Beckert |
| | Dr. Antonio Cerone |

# Acknowledgments

This thesis is the result of my work at University of Koblenz Artificial Intelligence Research Group. I want to thank my doctoral advisor Prof. Dr. Bernhard Beckert for the deep discussions about the topic of the thesis and uncountable valuable comments on early drafts of this work. I also want to thank Prof. Dr. Ulrich Furbach, and all other friends and colleagues from the AI Research Group. It was a pleasure to work with you. I want to thank especially those with whom I worked closely in various projects over the years, namely Bernd Thomas, Christian Wolff, Vladimir Klebanov, Pia Breuer, Markus Wagner, Niklas Henrich, and Thorsten Bormer. Dr. Roman Neruda provided me with the opportunity to spend part of my PhD studies at Czech Academy of Sciences.

Apart from my colleagues, I want to thank my parents Margret Bauer and Peter Beuster for supporting me in all my scientific (and not-so-scientific) endeavors. I am most grateful to my partner Eva-Maria Kahrau, who supported me in all the years.

## Abstract

This dissertation introduces a methodology for formal specification and verification of user interfaces under security aspects. The methodology allows to use formal methods pervasively in the specification and verification of human-computer interaction. This work consists of three parts. In the first part, a formal methodology for the description of human-computer interaction is developed. In the second part, existing definitions of computer security are adapted for human-computer interaction and formalized. A generic formal model of human-computer interaction is developed. In the third part, the methodology is applied to the specification and verification of a secure email client.



## Zusammenfassung

In dieser Dissertation wird eine Verfahrensweise für die formale Spezifikation und Verifikation von Benutzerschnittstellen unter Sicherheitsaspekten vorgestellt. Mit dieser Verfahrensweise können *beweisbar sichere* Benutzerschnittstellen realisiert werden. Die Arbeit besteht aus drei Teilen. Im ersten Teil wird eine Methodologie für die formale Beschreibung von Mensch-Maschine-Interaktion entwickelt. Im zweiten Teil werden gängige Computersicherheitskonzepte für die Mensch-Maschine-Interaktion angepasst und mit den im ersten Teil entwickelten Methoden formalisiert. Dabei wird ein generisches formales Modell von Mensch-Maschine-Interaktion erstellt. Im dritten Teil wird die Methodologie, die in den ersten beiden Teilen entwickelt wurde, an einem sicheren Email-Client als exemplarischen Anwendungsprogramm demonstriert.

# Contents

# Chapter 1

# Introduction

## 1.1 Goals and Structure

Off-the-shelf computer systems are regularly used for security-critical applications. In applications like home banking, the main task of the user's computer is to provide an interface to the computer system of his bank. The security of home banking and similar applications critically depends on the security of the interface provided to the user. Attackers actively try to exploit security weaknesses in graphical user interfaces of email programs and web browsers. This leads to an increasing demand for formal methods which are able to *guarantee* security in human-computer interaction.

While formal methods are used extensively in many fields of computer security, they are rarely used in Human-Computer-Interaction (HCI)—even for security critical systems. The reason is that HCI does not deal with the interaction of two machines but with the interaction of a machine and a human.

Countering human-computer interaction security threats by traditional software engineering methods suffers from two weaknesses: The first problem is that possible security holes may be overlooked. The second problem lies in the nature of human-computer interaction. Established methods for software specification, verification, and testing depend on explicit and exhaustive definitions of the interfaces and the behavior of software components. Since exhaustive descriptions of human behavior are generally impossible, rigorous formal methods are rarely used in human-computer interaction engineering. Most methods for describing human behavior in human-computer interaction are rather psychological and not well integrated in the software engineering process.

User models are routinely used in computer system usability studies. Such user models usually draw on psychological models of the user. They model the user's tasks, goals, motivations, etc. While this is essential under a usability point

of view, it makes a comprehensive formal modeling of the effects of user actions infeasible because complex psychological activities can be modeled to a limited extent only. From a usability point of view, this is not necessarily a severe drawback. To guarantee a certain level of usability, it suffices to give plausible evidence that an application's interface is usable, assuming certain goals and behaviors of the user. Security, however, requires a stricter notion of human-computer interaction. While a usability glitch in some dialog window may decrease the general usability of the application a bit, a security glitch can have more severe consequences. Even worse, a security glitch will encourage attackers to seek methods to actually exploit the glitch. The different view on the user and the different goals of usability and security make it possible and advisable to apply formal methods to security aspects of user interfaces with user models adapted to the particular needs of security. In this work we introduce such a formal methodology. Our methodology is based on the pervasive application of formal methods to the development and evaluation of human-computer interaction. It is able to *guarantee* that user interfaces do not contain security-critical errors.

The limits of the methodology introduced in this thesis lie in the general limits of formal methods for software engineering: Security is guaranteed only if the specification is an adequate representation of reality. While the formal specification of the behavior of computer systems can be achieved relatively easily, formal specification of user behavior is tricky. In the general case, we can not *know* that all users will always behave as modeled. We can, however make reasonable assumptions about user behavior, and we can provide explicit instructions for users. On the user's side, our methodology is based on simple, plausible, and explicit assumptions about user behavior. This allows to give an explicit description of the user behavior required to *guarantee* secure HCI.

In order to formally specify and verify the security of a user interface, it is necessary to bring together formal methods, human computer interaction, and computer security. All three of these are established fields of research. There are also works combining each two of the fields. Formal methods have been used to specify human computer interaction. User interfaces have been designed and evaluated under security aspects. System security has been treated with formal methods. In order to guarantee secure human-computer interaction by use of formal methods, all *three* fields have to be combined. This is the topic of this work. The structure of this work is as follows:

The formal methodology is defined in the first part. In the second part, a generic system model and a formal definition of human-computer interaction security is developed. Parts of the Common Criteria for Information Technology Security Evaluation (CC) (Common Criteria Evaluation Board (CCEB), 2006) are formalized. The methodology and security requirements developed in the first two parts are applied to a real-world application, a secure email client, in the third part.

The formal methodology introduced in the first part is generic for all applications and users. All kinds of application and user behaviors can be formally modeled with the methodology from this part of the thesis. The formal definitions of HCI security introduced in the second part are generic as well. A system designer can either choose a bottom-up approach and pick suitable formal security requirements from the formalized CC criteria defined in Chapter 10, or choose a top-down approach by designing the system in compliance with the formal definitions of HCI Confidentiality, Integrity, and Availability given in Chapter 11. In the third part of the thesis, the generic modeling methods and security requirement definitions are applied to a concrete application. While the third part of the thesis deals with one concrete application, the security measures developed in this part are applicable to similar applications, too. For all keyboard-driven applications with text output, the main event loop developed in Chapter 14 is applicable. The measures to guarantee confidentiality in Chapter 15 are generic for all applications with keyboard input and screen output. While the concrete definitions of desirable and undesirable states in the Chapter 16 depend on the actual application, the method to guarantee availability is generic.

**Part I** In the first part, the formal methods used in this work are introduced. In Chapter 4, we start with Input-Output Labeled Transition Systems (IOLTS) and Computational Tree Logic (CTL), the core formal methods used. Based on this, a formal method for user modeling is developed in Chapter 5. The user modeling methodology presented is based on the well-established GOMS methodology (John and Kieras, 1996). GOMS is extensively used for the modeling of user behavior. For our purposes, however, it has two weaknesses: A strict formal semantics is missing, and GOMS models the user behavior independently of the behavior of the system. Both of these short-comings are overcome in Chapter 5. In Section 5.1, we develop a formal semantics for GOMS models and illustrate it with an example. In Section 5.2, the example is completed by adding components representing the application and the user's assumptions about the application. Throughout the first part, a simple eVoting application serves as an example for the methodology.

For the pervasive verification of human-computer interaction, it is necessary to model HCI at all levels of detail. In Chapter 6, our approach is extended to hierarchical models. We show how the chosen modeling mechanism allows to model HCI from the highest to the lowest level while maintaining model sizes suitable for automated reasoning. This supports the *pervasive* description of HCI security and to prove security for all aspects of a user interface—from the pixel level up to high-level functionality of the user interface.

IOLTS and CTL are suitable methods to describe the concurrent behavior of

components, but they operate on abstractions of the actual program, not on the program itself. For the formal specification of computer programs on the level of individual procedures, different formal methods like Hoare logic (Hoare, 1969) are used. In order to make pervasive specification and verification of interactive applications possible, Hoare logic is integrated into our methodology in Chapter 7. Chapter 8 summarizes the result of the first part.

**Part II** Typically, HCI security requirements are informal and written for specific areas of applications. For example, the BSI's[1] protection profile for signature creation devices does not use formal methods to specify user interface requirements. The signature creation protection profile uses informal descriptions like "the data to be signed (DTBS) [has] to be displayed correctly"(SSCD-PP). In the second part of this work, we show how this kind of informal and specific requirements can be subsumed under generic and formal concepts. Formal definitions of human-computer interaction security criteria are developed and basic mental behaviors of the user (goal-orientation, mental representation of system states, etc.) are included in our formal methodology.

Generic models of the user, the computer system, and the processes running on the computer system are introduced in Chapter 9. The Common Criteria for Information Technology Security Evaluation (CC) (Common Criteria Evaluation Board (CCEB), 2006) are an international standard for computer security evaluation. In Chapter 10, they are analyzed in respect to user interface security. A set of core concepts is developed and formalized. Based on these core concepts, Common Criteria security criteria definitions are formalized. We show which combinations of criteria must be satisfied in order to guarantee secure human-computer interaction.

In Chapter 11, formal definitions of the basic security concepts *Confidentiality*, *Integrity*, and *Availability*, are developed for HCI. We show how these concepts relate to the Common Criteria core concepts developed in the previous chapter. With the results of part one and two, it becomes possible to pervasively specify and verify human-computer interaction under security aspects.

**Part III** Throughout the first two parts a simple eVoting application is used to demonstrate our methodology. In part three, the feasibility of our methodology is demonstrated on a real application. We apply our methodology to the specification and verification of a simple, text-based email client. An email client is a suitable exemplary application for a number of reasons. Email clients are one of the most popular applications on personal computers. The core feature of an email client

[1] Bundesamt für Sicherheit in der Informationstechnik / Federal Office for Information Security; Germany's government agency for information technology security

is to provide a user interface for displaying and editing email. For both of these core features user interface security is essential. Interpreting an email incorrectly, or sending an email with the wrong content or the wrong receiver poses security risks. Typical examples for this are phishing attacks, where the user thinks an email contains valid information from a legitimate source like his bank, while the actual source is an attacker and the data is malicious. The email client makes use of all basic types of user interface elements, and typical user behavior patterns are taken into consideration.

Chapter 13 describes the scenario of the email client application as part of the Verisoft project (Beuster et al., 2006). Based on the formal security requirements of *Confidentiality*, *Integrity*, and *Availability*, a specification of a secure email client satisfying the security requirements from Part II is developed in Chapters 14, 15, and 16.

## 1.2 Main Contribution

Each of the three parts constitutes a contribution to the field of formal methods for secure human-computer interaction:

**Part I** In the first part formal semantics for GOMS are developed, and GOMS is extended such that pervasive specification of human-computer interaction, including human error, becomes possible. The hierarchical approach to user interface and user behavior modeling introduced in this part allows a pervasive formal treatment of human-computer interaction on all levels, from a most abstract view of general application behavior and user's intentions and goals, down to lowest-levels of application and user behavior. The integration of Hoare logic for specification of procedures allows for the complete formal specification and verification of program behavior, while at the same time it is possible to describe the abstract concurrent behavior of components by IOLTS and CTL.

**Part II** The second part presents a systematic adaptation of the core principles of computer security to human-computer interaction security and its translation from informal definitions to formal definitions suitable for automated reasoning. Parts of the Common Criteria for Information Technology Security Evaluation (CC) (Common Criteria Evaluation Board (CCEB), 2006) are formalized for user interface security.

**Part III** In the third part, we present a prototypical application that has been pervasively specified and verified under user interface security considerations. In

contrast to other areas of application of formal methods, the prototypical application is not highly specialized for a specific security or safety relevant area. It is a prototype for applications typically used on a daily basis by end-users on open networks like the Internet. Proofs are given showing which combinations of components and behavioral traits satisfy the security criteria developed in part two. The results of the third part provides system builders with a realistic set of building blocks for the specification and evaluation of user interfaces for security critical applications.

We provide a formal methodology for the pervasive specification and verification of human-computer interaction based on and derived from the fundamental principle of computer security. Our work contributes to computer science by developing new methods for formal specification of secure user interfaces, and by formalizing security requirements of user interfaces. These areas are highly relevant for practical software development. Formal methods for security of user interfaces are applicable to a large number of applications, e.g. e-banking applications, ATMs, email clients, but also safety critical systems like medical devices. A number of governmental organizations have issued security criteria catalogs and for applications deployed in security sensitive areas. These standards are adapted by private industry as well. Since neither formal methods to describe security aspects of user interfaces were available until now, nor formal criteria to evaluate security of interfaces, these catalogs make little use of formal methods and in general do not require verification at all. In this work, we get user interface security into the realm of formal specification and verification.

Parts of Chapters 4, 5 and 6 have been published in Beckert and Beuster (2006b). Parts of Chapters 11 and 13 have been published in Beuster et al. (2006). Parts of Chapter 14 have been published in Beckert and Beuster (2006a, 2007). Parts of Chapter 15 have been published in Beckert and Beuster (2004). Parts of Chapter 16 have been published in Beckert et al. (2007).

# Chapter 2

# Related Work

We build upon work on formal methods for developing computing systems, human-computer interaction (HCI) research, and secure system design. Abowd et al. (1989) and Jain (1994) give a survey of formal languages for the description of user interfaces. More overviews are given in two (different) books called *Formal Methods in Human-Computer Interaction* (Harrison, 1990; Palanque and Paternò, 1998).

There are three main areas of research in formal methods for human-computer interaction. The first area develops and analyzes formalisms for the description of interfaces and human-computer interaction. The second area investigates the integration of user interface components in the formal software development process. The third area is the development of tools for formal description and development of interfaces. In the following, we review existing approaches in these three areas.

## 2.1 Formalisms for Describing HCI

We distinguish between "black box" and "white box" methods for describing interfaces. "Black box" methods describe the behavior of a component by its input and output interfaces. When describing HCI, the output interface is typically the screen and the input interface consists of a keyboard and a mouse. The internal structure of a component is not relevant from this point of view. "White box" methods describe the internal structure of the component.

A good formalism for user interface design supports multiple levels of abstraction both for the black box and for the white box view. For the black box view, it should be possible to describe user interface elements on different levels of aggregation. For screen output, this ranges from pixel-level description of screen elements to aggregated descriptions of sets of widgets. For keyboard input, it should be possible to capture single keystrokes and their timings as well as

aggregated sets of keystrokes representing commands. Another important feature of techniques suitable for modeling human computer interaction is the ability to deal with concurrency.

An early contribution to formal methods for the description of user interaction is the PIE model, developed by Dix and Runciman (1985). PIE and its more recent variations (e.g. Dix and Abowd (1996)) allow to describe the interaction of the system and a user formally, but they focus on describing the computer system's side of the interaction. In PIE, the behavior of a user interface is described by a sequence of commands (issued by the user) leading to a sequence of effects. In this model, system behavior is defined as a function from commands issued by the user to effects produced by the system. In case of a text-based user interface, the input is a sequence of keystrokes and the output are characters displayed on the screen.

PIE and similar formalisms put an emphasis on describing the I/O behavior of a computer system and are suitable for automated reasoning, e.g. with model checkers. Rushby (2002) uses model checking in order to detect potential discrepancies between system behavior and the mental models of system users. The main weakness of PIE is that it focuses on the behavior of the computer system. It does not provide advanced mechanisms for user modeling.

Carr (1997) introduces Interaction Object Graphs (IOG), an extension of statecharts for modeling elements of graphical user interfaces and their interactions. IOG allow a description both on the pixel-level and on an aggregated level. IOG focuses on graphical user interfaces, and the language used to describe them is directly executable. The formalism of IOG allows basic reasoning tasks like testing for reachability of all states. Sucrow (1997) uses graph grammars to describe graphical user interface elements. Changes in the GUI are modeled by re-write rules. The main weakness of both approaches is that they are primarily languages for formal specification of user interfaces, but not for the formal description and analysis of human-computer interaction.

Palanque et al. (1995) use hierarchical Petri nets to combine user and system models of interactive systems. Berstel et al. (2005) developed "Visual Event Grammars" (VEG), a formal method for the specification and validation of graphical user interfaces. They describe complex graphical user interface as communicating automata. Interactive Cooperative Objects (ICO) (Palanque and Bastide, 1994) are a specialization of High Level Petri Nets (HLPN) for user interface description. The approach by Palanque et al. is similar to the approach presented in this thesis, but in contrast to Palanque we do not use Petri nets for modeling, but IOLTS.

In a number of works, formal specification methods like Z have been applied to user interface design. One of the first formal specifications of interactive components was the specification of a text editor in Z presentd in *Formal specification*

*of a display editor* by Sufrin (1982). Based on Sufrin's specification, Booth and Jones (1994) implemented an editor in the Miranda functional programming language. Goldson (2000) and Hussey and Carrington (1998) provide more case studies in using Z for user interface specification. In Part III, we present the specification of a secure email client including an interactive editor.

Most formalisms use variants of state transition diagrams for the white box model and process algebras for the black box model of interactive systems. Exceptions include XTL (Brun, 1998), which uses temporal logics, hybrid methods and non-standard methods like MAL and interactors (Palanque and Paternò, 1998). Process algebras for the black box view are used by Cabrera et al. (1995) and Kuhn and Frank (1991). Communicating Sequential Processes (CSP) (Smith and Duke, 1999) and LOTOS (van Eijk et al., 1989) have also been used to specify human computer interactions.

While PIE and similar formalisms put an emphasis on describing the I/O behavior of a computer system and are suitable for automated reasoning (e.g., with model checkers), other approaches like Task Knowledge Structures (TKS) (Hamilton, 1996), (Extended) Task Action Grammar ((E)TAG) (de Haan, 2000), and Goals Operators Methods Selection-rules (GOMS) (John and Kieras, 1996) focus on providing cognitive models of the user. TKS provides an explicit representation of the cognitive model of the user. TAG allows a precise formal description of the user actions, the user's knowledge and the user's internal representation of the system (what the user thinks about the system.) ETAG is an extension of Task-Action-Grammar. ETAG's formal model represents the knowledge of the user about the user interface. In ETAG, the interface provided by the machine to the user is described as a "User's Virtual Machine" (UVM). It uses object oriented design with precondition/action/postcondition style specifications of actions. The mental model used by ETAGs is restricted in the way that it does not have an explicit model of the user's internal mental states. It does, however, make assumptions about the user's knowledge. For our approaches, ETAG is too high-level, because it "does not describe the details of the presentation of information on the display screen and the specific knowledge of particular users and the strategies they use" (de Haan, 1995). Another disadvantage of ETAG is that "it does not address the presentation interface. [...] elements of the presentation interface [...] are named or mentioned, but these are only included insofar they are needed to completely describe the non-graphical aspects of the interface" (de Haan, 1995).

A general weakness of formal HCI methods like TKS and (E)TAG is that they require detailed models of the user behavior in order to model the interaction between a computer system and a user. While computer systems can (and should) be formally specified, a formal user model is always based on assumptions about the user which may or may not be true. The approach presented in this thesis requires minimal assumptions about the user.

Harrison (1990) develops the concept of "State Display Conformance" which is closely related to the integrity requirement developed in Chapter 11 of this work. It should be noted that Grudin's argument *against* user interface consistency requirements (Grudin, 1989) does not apply to the work presented here. He argues that consistency defined as having similar user interface elements for similar functionality can not be generalized, because similarity depends on context. Our work however does not address consistency within a user interface, but consistency between a user's mental representation of a system state and the actual system state.

## 2.2 Tools

Most tools for formal user interface design support declarative methods. Tools like MASTERMIND (Browne et al., 1997), TADEUS (Schlungbaum and Elwert, 1995) and VEG (Berstel et al., 2005) aid the user in the formal description of user interfaces. MASTERMIND focuses on static description of user interface elements, while TADEUS uses graph notation to describe the behavior of user interfaces. TADEUS is embedded in a method for developing applications based on the separation of the functional core of an application from its user interface. MASTERMIND and VEG do not only support the formal design of user interface, but also proves about properties of user interfaces by model checking.

## 2.3 Design Methods

PAC-Amodeus (Calvary et al., 1997) is a typical design method for applications with user interfaces. It separates the functional parts of a software from the interactive components. User interface elements are represented as agents. A *dialog controller* provides the interface between the user interface part of a system and the functional part. Although a system design like this seems to be a natural view when constructing a software system from components, it has major drawbacks from a security point of view. Security usually concerns all aspects of a system, and the security of an element from the functional core may depend directly on properties of the user interface. Even worse, security may depend on the interaction of different parts of the system, both from the functional core and the user interface.

## 2.4 Summary

Most existing formal methods for specification of user interfaces describe user interfaces on an abstract, high level. They define properties of a (most times fixed) sets of widgets, and the interactions of these widgets. In general, they do not take (potentially undesirable) effects caused by interaction of different user interface elements into account. This is a problem, because data may come from non-trustworthy sources. We want to show that it is not possible to bring the user interface into a configuration where the user is deceived about the state of the system. For example, a common trick by advertisers on the World Wide Web is to mimic the appearance and behavior of system status windows in order to get the attention of the user. This trick works by using low-level functionality (showing a bitmap supplied by the attacker) to mimic high level functionality (making the bitmap look like a status window).

Therefore, we need a modeling language suitable for analysis of interactions between "high level" properties ("the widget showing the system state should always be on top of all other widgets") and "low level" properties ("The colors and fonts used to show the warning message should be legible.") of the interface.

## 2.5 GOMS

GOMS is similar to (E)TAG. While TAG describes user activities in categories of "tasks" and "actions", GOMS breaks HCI down into *Goals*, *Operators*, *Methods*, and *Selection rules* (John, 1995). Goals are the tasks the user wants to accomplish. In order to do so, he issues commands to the computer system via operators. This can be text input, mouse movements, etc. Methods are strategies to accomplish (sub-)tasks available to the user, e.g. moving a text block, coloring a box, etc. Methods can be further decomposed into sequences of sub-goals and operators utilized to achieve the sub-goal. Selection rules must be employed by the user if there is more than one method to accomplish a task.

GOMS is a well established formal method for the description of HCI from a user's perspective. It is based on the solid ground of psychological research. GOMS, like most other methods for user modeling, are geared towards studying usability. It has some weaknesses that are particularly problematic in the context of security: GOMS has no means to describe concepts like user fatigue, individual difference between users, etc. (Olson and Olson, 1995).

There are different flavors of GOMS. KLM-GOMS (Keystroke-Level-Model GOMS) describes user activity on the lowest level (John, 1995). As the name suggests, user behavior is described by measuring the time it takes an experienced user to press keys, move the mouse to certain areas of the screen, etc. In KLM-

GOMS, the user executes a fixed sequence of operators in order to achieve a task. There are no selection rules. KLM-GOMS is used to measure the time it takes an experienced user to accomplish a known task.

> "The Rationality Principle asserts that users will develop methods that are efficient, given the structure of the task environment (i.e., the design of the system) and human processing abilities and limitations. Thus, human activity with a computer system can be viewed as executing methods to accomplish goals, and because humans strive to be efficient, these methods are heavily determined by the design of the computer system. This means that the user's activity can be predicted to a great extent from the system design. Thus, constructing a GOMS model based on the task and the system design can predict useful properties of the human interaction with a computer."(John and Kieras, 1996, p. 10)

GOMS is oriented at psychological analysis of user behavior and timed measurement of user activity. A major weakness of GOMS is that it is limited to sequential user plans, and that it does not provide means to generate application specifications from user models. This rather Tayloristic approach to HCI has drawn critique for not being mentally adequate and not taking inexperienced users and users who may make mistakes from time to time into account. Newell (1994), one of the creators of GOMS has developed an advances cognitive modeling methodology, called SOAR. The aim of SOAR is to create an architecture suitable to model all kinds of intelligent behaviors. Since our goal is to model the user as simple and as general as possible, the sophisticated modeling methods provided by SOAR are not required. We base our user modeling technique on GOMS.

CMN-GOMS augments KLM-GOMS with selection rules and sub-goals. We use the CMN-GOMS, because selection rules are essential to our approach, Moreover CMN-GOMS (John and Kieras, 1996) allows to describe user models hierarchically. This is an important property for modeling a user interface under security aspects because of the large variety of errors in human-computer interaction. Some of these errors are on a very low level (for example, the user may push the mouse button twice instead of once), while others are on a very high level of abstraction (e.g., the user may misinterpret the meaning of an error message). A hierarchical modeling mechanism allows to model all kinds of errors within one formalism. CMN-GOMS models are semi-formal. We provide formal semantics for CMN-GOMS models in Chapter 5. The formal CMN-GOMS model is augmented by formal models of the application and formal models of the user's assumptions about the application. With a formal definition of secure human-computer interaction, this allows to determine the security of a user interface by automated reasoning.

Another advantage for our purpose is that GOMS description method is very close to the State Transition Diagrams that we use to formalize user behavior:

> "[...] CMN-GOMS is based on two of the MHP 'Principles of Operation', the *Rationality Principle* and the *Problem Space Principle* [...]. The Problem Space Principle postulates that a user's activity can be characterized as applying a sequence of actions, called *operators*, to transform an initial state into a goal state."(John and Kieras, 1996, p. 10)

# Part I

# Foundations

# Chapter 3

# Overview of Part I

The application of formal methods to problems of secure human-computer interaction requires a common foundation for the formal modeling of human-computer interaction. A common formal language is required as the base for the description of human behavior, application behavior, human-computer interaction, and security criteria definition. This formal language must allow an adequate modeling of the components involved in human-computer interaction, namely the user(s), the application(s), and the channels of communication between them. In order to keep our methodology as generic as possible, the language should allow for "block box" modeling, i.e. it should be possible to describe components by the messages sent and received, without having to know about the internal structure of the components. Also, the formal methods should be suitable for automated reasoning. Input-Output Labeled Transition Systems (IOLTS) as a suitable method are introduced in Chapter 4.

In Chapter 5, the language defined in Chapter 4 is used to develop a formal methodology for the description of user behavior. The user modeling methodology presented in this chapter is based on the well-established GOMS methodology (John and Kieras, 1996). GOMS describes user behavior in the categories of the user's *Goals*, the *Operators* available to the user, the *methods* employed by the user, and the *Selection Rules* used by the user to choose if more than one method is available to achieve a goal. GOMS is extensively used for the modeling of user behavior. For our purposes, however, it has two weaknesses: Strict formal semantics are missing, and GOMS models the user behavior independently of the behavior of the system. Both of these short-comings are overcome in Chapter 5.

Chapter 6 extends our methodology to hierarchical models. With hierarchical models, it becomes possible to model human-computer interaction on an arbitrary level of details, ranging from a coarse-grained view of interaction of abstract concepts like the user's general goals and the logical structure of the application, down to a low-level model of technical details like the visualization of individual

elements on the screen. An abstraction method for hierarchical models is introduced in order to allow automated reasoning even for highly complex models with large state spaces.

# Chapter 4

# IOLTS and CTL

## 4.1 IOLTS

As we have seen in Chapter 2, most work on formal methods for user interface specification and human-computer interaction makes use of graph-based methods or of methods that can be reduced to graph-based formalisms. We follow this line of work and base our methodology on Input-Output Labeled Transition System (IOLTS) and Computation Tree Logic (CTL). Labeled Transition Systems are commonly used to define the semantics of formal methods like process algebras. Tools like model checkers are usually based on LTS, where more complex formalisms are translated to LTS in a pre-processing step. The extension of LTS to IOLTS allows to describe behavior of LTS by traces of input- and output-symbols. We describe properties of IOLTS in Computational Tree Logic (CTL) formulae. Automated tools like the NuSMV model checker are able to check if CTL formulae are satisfied by models given as LTS. In Chapter 5, we develop a methodology for the formal description of and reasoning about GOMS that make use of Input-Output Labeled Transition Systems (IOLTS) and Computational Tree Logic (CTL). Below, we define these concepts and some related notions used in the next chapters.

**Definition 4.1** (LTS)**.** *A* Labeled Transition System *(LTS) is a tuple* $L = (S, \Sigma, s_0, \rightarrow)$ *where* $S$ *is a set of* states, $s_0 \in S$ *is an* initial state, $\Sigma$ *is a set of* labels, *and* $\rightarrow \subseteq S \times \Sigma \times S$ *is a* transition relation. *We use the notation* $p \xrightarrow{\sigma} q$ *for* $(p, \sigma, q) \in \rightarrow$. *The special label* $\varepsilon \in \Sigma$ *indicates a silent transition from one state to the next.*

**Definition 4.2** (IOLTS)**.** *An* Input-Output Labeled Transition System *(IOLTS) is an LTS* $L = (S, \Sigma, s_0, \rightarrow)$ *with* $\Sigma = \Sigma? \cup \Sigma!$. *We call* $\Sigma?$ *the* input alphabet *and* $\Sigma!$ *the* output alphabet.

We use state transition diagrams to visualize IOLTS. An example is shown in Figure 4.1.

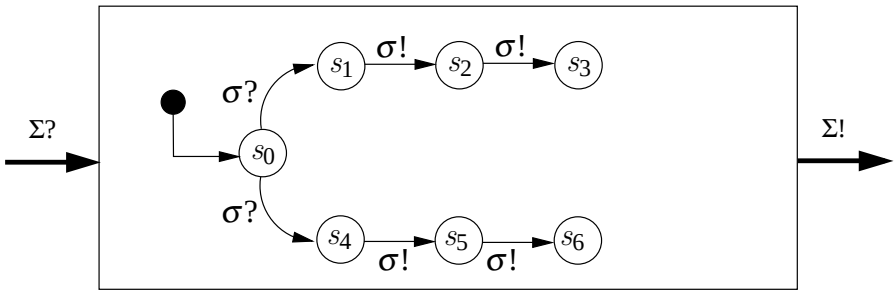


Figure 4.1: State Transition Diagram representation of an IOLTS.

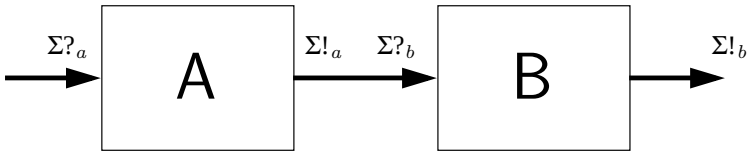


Figure 4.2: Composition of two IOLTS.

A *linear composition* is the concatenation of two IOLTS $L_a$ and $L_b$ where the output of $L_a$ is input for $L_b$ as shown in Figure 4.2:

**Definition 4.3** (Linear Composition)**.** *Let* $L_a = (S_a, \Sigma_a, s_{0\,a}, \rightarrow_a)$, $L_b = (S_b, \Sigma_b, s_{0\,b}, \rightarrow_b)$ *be two IOLTS with* $\Sigma?_a \cap \Sigma?_b = \{\}$ *and* $\Sigma!_a \cap \Sigma!_b = \{\}$*. The* composition $(L_a.L_b) = (S, \Sigma, s_0, \rightarrow)$ *of* $L_a$ *and* $L_b$ *is defined as:*

$$
\begin{aligned}
S &= S_0 \times S_1 \\
\Sigma? &= \Sigma?_a \\
\Sigma! &= \Sigma!_a \cup \Sigma!_b \\
s_0 &= (s_{0\,a}, s_{0\,b}) \\
\rightarrow &= \{((s_a, s_b), \sigma, (s'_a, s_b)) \mid s_a \xrightarrow{\sigma}_a s'_a \text{ with } \sigma \in \Sigma?_a \cup \Sigma!_a\} \cup \\
&\quad \{((s_a, s_b), \sigma, (s_a, s'_b)) \mid s_b \xrightarrow{\sigma}_b s'_b \text{ with } \sigma \in \Sigma?_b \cup \Sigma!_b\} \cup \\
&\quad \{((s_a, s_b), \varepsilon, (s'_a, s'_b)) \mid s_a \xrightarrow{\sigma}_a s'_a \text{ and } s_b \xrightarrow{\sigma}_b s'_b \text{ with } \sigma \in \Sigma!_a \cap \Sigma?_b\}
\end{aligned}
$$

Intuitively, a linear composition acts like incoming labels are processed by the first IOLTS producing some output, which in turn is processed by the second IOLTS. In common definitions of composition, the output alphabet of the first IOLTS must be identical to the input alphabet of the second IOLTS, and the output

of the first IOLTS is completely consumed by the second IOLTS, i.e. output of the first IOLTS does not show up in the output of the composed component. This alternative definition of linear composition is not suited for our purposes, because the output of the first IOLTS is sometimes processed by more than one other IOLTS and must therefore be preserved. A small example is given in Figures 4.3 and 4.4.

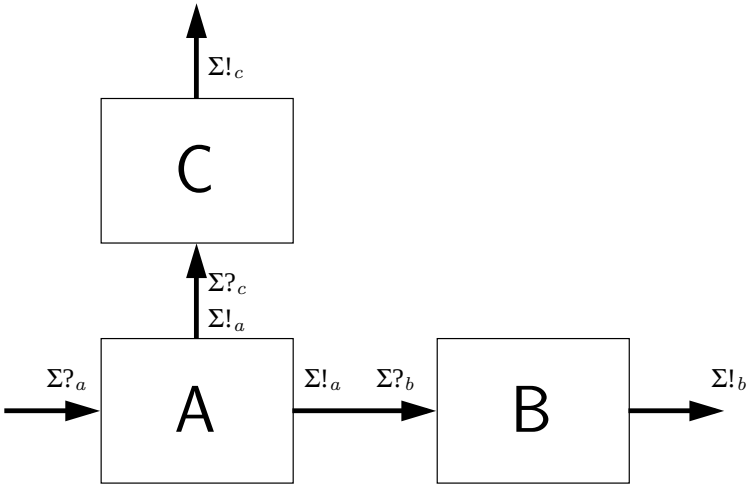


Figure 4.3: Before linear composition of A and B.

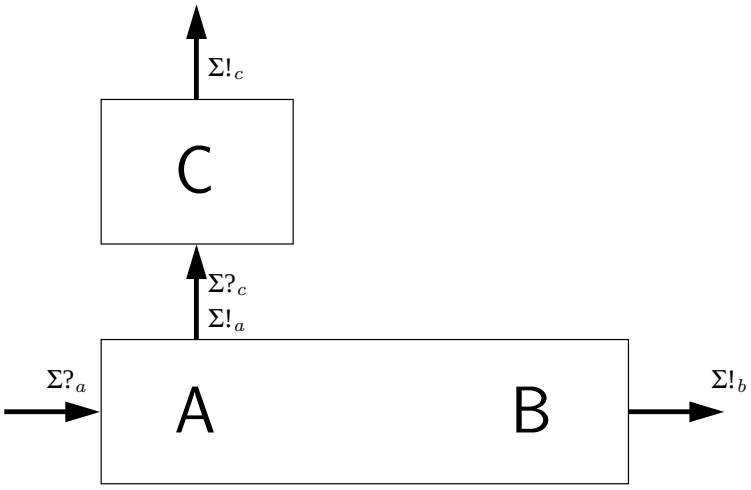


Figure 4.4: After linear composition of A and B.

Our definition takes this into account by adding $\{((s_a, s_b), \sigma, (s'_a, s_b)) \mid s_a \xrightarrow{\sigma}_a s'_a \text{ with } \Sigma!_a\}$ to the transitions of the composed IOLTS. In difference to common

definitions of linear composition, in our definition the output from the first IOLTS is forwarded and becomes output of the composed component. Chapter 6 makes extensive use of this kind of composition.

Often, components are combined by *parallel* composition. In parallel composition, the output of $L_a$ serves as input for $L_b$, and the output of $L_b$ serves as input of $L_a$ (see Figure 4.5).

**Definition 4.4** (Parallel Composition)**.** *Let* $L_a = (S_a, \Sigma_a, s_{0a}, \rightarrow_a)$ *and* $L_b = (S_b, \Sigma_b, s_{0b}, \rightarrow_b)$ *be IOLTS.*

*We assume the input and output alphabets of* $L_a$ *and* $L_b$ *to consist of internal and external subsets, where the internal input is denoted with* $\Sigma?I$*, the external input with* $\Sigma?E$*, the internal output with* $\Sigma!I$*, and the external output with* $\Sigma!E$*. And we require that these subsets are chosen such that* $\Sigma!I_a = \Sigma?I_b$ *and* $\Sigma!I_b = \Sigma?I_a$*.*

*The* parallel composition $(L_a \parallel L_b) = (S, \Sigma, s_0, \rightarrow)$ *of* $L_a$ *and* $L_b$ *is defined as:*

$$
\begin{aligned}
S &= S_0 \times S_1 \\
\Sigma? &= \Sigma?E_a \cup \Sigma?E_b \\
\Sigma! &= \Sigma!E_a \cup \Sigma!E_b \\
s_0 &= (s_{0a}, s_{0b}) \\
\rightarrow &= \{((s_a, s_b), \sigma, (s'_a, s_b)) \mid s_a \xrightarrow{\sigma}_a s'_a \text{ with } \sigma \in \Sigma?E_a \cup \Sigma!E_a \cup \Sigma I_a\} \cup \\
&\quad \{((s_a, s_b), \sigma, (s_a, s'_b)) \mid s_b \xrightarrow{\sigma}_b s'_b \text{ with } \sigma \in \Sigma?E_b \cup \Sigma!E_b \cup \Sigma I_b\} \cup \\
&\quad \{((s_a, s_b), \varepsilon, (s'_a, s'_b)) \mid s_a \xrightarrow{\sigma}_a s'_a \text{ and } s_b \xrightarrow{\sigma}_b s'_b \text{ with} \\
&\qquad\qquad \sigma \in \Sigma!I_a \cup \Sigma!I_b\}
\end{aligned}
$$

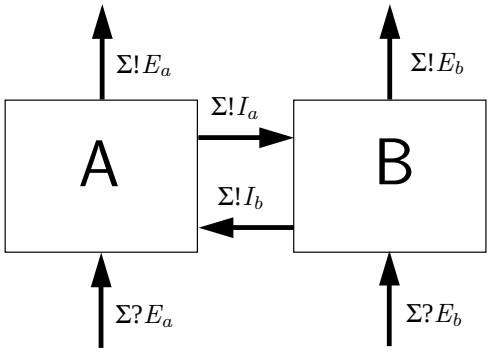


Figure 4.5: Parallel composition of IOLTS.

Below, we additionally use a variant of IOLTS called Symmetric Input Output Labeled Transition Systems (SIOLTSs), where each transition produces both an input and an output symbol. SIOLTS play an important role in the integration

of IOLTS-based specification and specifications in Hoare logic. In Chapter 7, we associate the labels of SIOLTSs with procedure calls, the input symbols with input variables, and the output symbols with output variables of the procedure calls.

**Definition 4.5.** *An LTS* $L = (S, \Sigma, s_0, \rightarrow)$ *with* $\Sigma = (\Sigma? \times \Sigma!)$*, for an* input alphabet $\Sigma?$ *and an* output alphabet $\Sigma!$*, is called* Symmetric Input Output Labeled Transition System *(SIOLTS).*

The input/output behavior of a component is described by *traces*, which are (possibly infinite) sequences of elements from the alphabet $\Sigma$, and *paths*, which are corresponding sequences of states.

**Definition 4.6** (Traces and Paths)**.** *Let* $L = (S, \Sigma, s_0, \rightarrow)$ *be an IOLTS. Then, a* path *is a sequence* $\langle s_0, s_1, \ldots \rangle$ *of states from* $S$ *with* $s_i \rightarrow s_{i+1}$ *for all* $i \geq 0$*. A* trace *(of* $L$*) is a sequence* $\langle \sigma_0, \sigma_1, \ldots \rangle$ *of elements of* $\Sigma$ *such that there is a path* $\langle s_0, s_1, \ldots \rangle$ *with* $s_i \xrightarrow{\sigma_i} s_{i+1}$ *(*$i \geq 0$*). Given a path* $c = \langle s_0, s_1, \ldots \rangle$*, by* $c^i$ *we denote the sub-path of* $c$ *starting at* $s_i$*.*

We describe properties of components in temporal logic. Our choice of temporal logic is based on two criteria: The logic of choice must be expressive enough for the description of security properties of HCI. Tools for automated reasoning (e.g. model checking) should be available. Computational Tree Logic (CTL) satisfies these requirements. Model checkers typically support CTL and Linear Temporal Logic (LTL). In this work, we have used the free model checker NuSMV 2 (Cimatti et al., 2002). LTL is not suitable for the security properties formalized in this thesis. It does not allow existential quantification over paths. As we will see in Chapters 10 and 11, existential quantification over paths is required to formalize parts of the common criteria and to formalize availability requirements. NuSMV 2 and other model checkers support propositional CTL. However, for the definition of security requirements and for the specification of component it is more convenient to use first-order logic. For example, we want to quantify over all users of a system, or all messages send by an application. In this thesis, we assume domains are finite. This makes it possible to treat FO-CTL like propositional CTL. See (Gilmore, 1960) for reduction of finite domain FOL to PL (propositionalization).

**Definition 4.7** (FO-CTL)**.** *Let* $P$ *be a set of* $n$*-ary relation symbols,* $V$ *a set of variables,* $C$ *a set of constants, and* $F$ *a set of functions. The set of terms* $\tau$ *is defined as*

$$\tau ::= c \mid v \mid f(t_1, \ldots, t_n)$$
*with* $c \in C$*,* $v \in V$*,* $f \in F$*, and* $t_1, \ldots, t_n \in \tau$

*FO-CTL formulae $\phi$ are constructed inductively by:*

$$\phi ::= p(t_1, \ldots, t_n) \mid \phi \vee \phi \mid \phi \wedge \phi \mid \neg\, \phi \mid \forall x.\phi \mid \exists x.\phi \mid \boldsymbol{E}\psi \mid \boldsymbol{A}\psi$$
*with $p \in P$, $x \in V$, and $t_1 \ldots t_n \in \tau$*

$$\psi ::= \boldsymbol{X}\phi \mid \phi \boldsymbol{U} \phi \mid \boldsymbol{G}\phi \mid \boldsymbol{F}\phi$$

Intuitively, $\mathbf{X}\phi$ means that $\phi$ holds in the next step, $\phi\mathbf{U}\phi$ means that $\phi$ holds from now on until $\phi$ holds, $\mathbf{G}\phi$ means that $\phi$ holds forever and $\mathbf{F}\phi$ means that $\phi$ will hold eventually. $\mathbf{E}\psi$ means that there exists a path where $\psi$ holds, and $\mathbf{A}\psi$ means that $\psi$ holds on all paths. For example, $\mathbf{AG}\,\mathit{likes}(\mathit{Bob}, \mathit{Soccer})$ means that whatever the future may be, Bob will always like soccer, and **EG** *likes(Bob, Soccer)* means that it is possible that in the future Bob will always like soccer.

Now, we can use IOLTS to interpret FO-CTL formula—in combination with valuation functions $\lambda$ from the set of states of an IOLTS to the set of interpretations over a domain. In the following chapters, we use IOLTS to model the high-level behavior of users and applications. We assume the domain is constant and finite.

**Definition 4.8** (IOLTS Semantics)**.** *Given an IOLTS $L = (S, \Sigma, s_0, \rightarrow)$, a domain $D$, and a set of interpretations $I$ a* valuation *$\lambda$ is a mapping from $S$ to $I$. $L, \lambda, c_0 \models \phi$ denotes that $\phi$ holds in state $c_0$ with valuation function $\lambda$. $L, \lambda, x \models \phi$ denotes that $\phi$ holds for all paths $x = \langle c_0, c_1, \ldots \rangle$ with valuation function $\lambda$. $\lambda$ is defined inductively as as follows:*

| | |
|---|---|
| $L, \lambda, c_0 \models p(t_1, \ldots, t_n)$ | *if* $(I(t_1), \ldots, I(t_n)) \in I(p)$ *with* $I = \lambda(c_0)$ |
| $L, \lambda, c_0 \models p(t_1, \ldots, t_n)$ | *if* $(I(t_1), \ldots, I(t_n)) \in I(p)$ *with* $I = \lambda(c_0)$ |
| $L, \lambda, c_0 \models \neg\, \phi$ | *if not* $L, \lambda, c_0 \models \phi$ |
| $L, \lambda, c_0 \models \phi \wedge \psi$ | *if* $L, \lambda, c \models \phi$ *and* $L, \lambda, c_0 \models \psi$ |
| $L, \lambda, c_0 \models \phi \vee \psi$ | *if* $L, \lambda, c_0 \models \phi$ *or* $L, \lambda, c_0 \models \psi$ |
| $L, \lambda, c_0 \models \forall x.\phi$ | *if* $L, \lambda, c_0 \models \phi_{[x/y]}$ *for all* $y \in D$ |
| $L, \lambda, c_0 \models \exists x.\phi$ | *if* $L, \lambda, c_0 \models \phi_{[x/y]}$ *for at least one* $y \in D$ |
| $L, \lambda, x \models \phi$ | *if* $L, \lambda, c_0 \models \phi$ |
| $L, \lambda, x \models \boldsymbol{A}\phi$ | *if* $L, \lambda, x \models \phi$ *for all paths* $x$ *in* $L$ *starting with* $c_0$ |
| $L, \lambda, x \models \boldsymbol{E}\phi$ | *if* $L, \lambda, x \models \phi$ *for at least one path* $x$ *in* $L$ *starting with* $c_0$ |
| $L, \lambda, x \models \boldsymbol{X}\phi$ | *if* $L, \lambda, x^1 \models \phi$ |
| $L, \lambda, x \models \phi \boldsymbol{U} \psi$ | *if (a)* $L, \lambda, c_0 \models \psi$ *or* |
| | *(b) there is some* $i \geq 1$ *s.t.* $L, \lambda, x^i \models \psi$ |
| | *and* $L, \lambda, x^k \models \phi$ *for all* $0 \leq k < i$ |
| $L, \lambda, x \models \boldsymbol{G}\phi$ | *if* $L, x^i \models \phi$ *for all* $i \geq 0$ |
| $L, \lambda, x \models \boldsymbol{F}\phi$ | *if* $L, x^i \models \phi$ *for some* $i \geq 0$ |

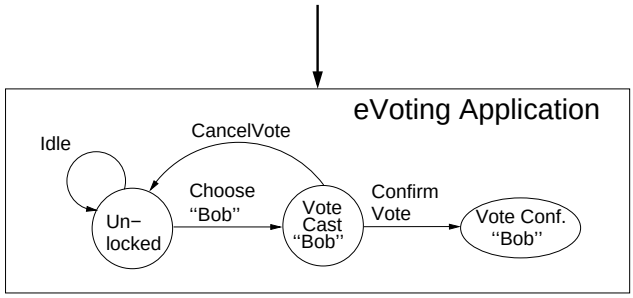


Figure 4.6: Application Model for the eVoting example (basic version).

We use an eVoting application as a running example throughout this thesis. The user is asked to select a candidate. After choosing, the eVoting application asks the user to confirm his vote. If he confirms, the voting process finishes. If he cancels, he can change the vote. An IOLTS modeling the simplest version of this application is shown in Figure 4.6. "Bob" is the only candidate in the example. A user operating the application is modeled in Figure 4.7.

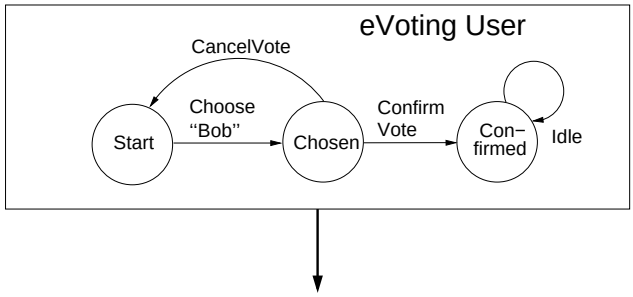


Figure 4.7: User Model for the eVoting example (basic version).

As an example we evaluate if it is possible that the final state "vote confirmed" is never reached. First, we give a valuation function $\lambda$ for the IOLTS shown in Figure 4.6 with $\lambda(s) = \{p \mid (\lambda(s))(p) = \text{true}\}$. In order to formalize the desired property, we only need an atomic proposition *final* which holds in state "Vote_Confirmed_Bob":

$$\begin{aligned} \lambda(\text{Locked}) &= \varnothing \\ \lambda(\text{Unlocked}) &= \varnothing \\ \lambda(\text{Vote\_Cast\_Bob}) &= \varnothing \\ \lambda(\text{Vote\_Confirmed\_Bob}) &= \{\textit{final}\} \end{aligned}$$

The requirement that a final state is always reached is defined in CTL as **AF***final*. A refutation of this proposition using NuSMV is given in Appendix A.

In the refutation, the user always cancels his vote, votes again for "Bob", cancels the vote, and so on. It is, however, possible to reach a final state: **EF***final*. Even more, it is always possible to reach a final state: **AGEF***final*.

## 4.2 User and Application Models

IOLTS as shown in Figure 4.8 are used to model the behavior of applications. We use FO-CTL formulae to describe properties of the applications. In order to develop a generic methodology for the description of security properties of applications, we use some pre-defined predicates in all our models. In the following, we give a list of these predicates. Each predicate is accompanied with a short explanation. The reader is referred to Chapters 10 and 11 for an in-depth discussion of these predicates.

In a number of situations it is necessary to refer explicitly to the application state as given in the IOLTS. For this, we define predicates with the same names as the states of the IOLTS which hold iff the IOLTS is in the corresponding state:

**Definition 4.9** (State Predicate)**.** *Let $L = (S, \Sigma, s_0, \rightarrow)$ be an IOLTS. Let $\lambda$ be a valuation function. The model $(L, \lambda)$ contains* state predicates *if the valuation function $\lambda$ has the following properties:*

$$\begin{array}{ll} L, \lambda, s & \models \mathit{state}(s) \\ L, \lambda, s' & \not\models \mathit{state}(s) \quad \mathit{if}\ s \neq s' \end{array}$$

In Section 11.1 security requirements for applications are based on special properties of certain states in the application model. We distinguish between four kinds of application states. *Success* states are the states where the user has achieved his goal. *Fatal* states are states undesirable for the user. The decisions about which states are the success states and which states are fatal are part of the model. The definitions of the other two kinds of states, *critical* states and *safe* states, follow from the definition of *success* and *fatal* states. If at least one transition from a state immediately leads into a *fatal* state then it is a *critical* state. All states neither *critical*, nor *success*, nor *fatal*, are *safe* states[1].

We define two predicates for each of the special states. States *success, fatal, critical, safe* hold iff the IOLTS is in the respective state. Predicates *success*$(s)$, *fatal*$(s)$, *critical*$(s)$, *safe*$(s)$ hold if IOLTS state $s$ is of type success, fatal, critical, safe, respectively.

[1] Depending on the application, one may want to extend the definition of *fatal* states as follows: If from a given state no *success* state is reachable, then it is a *fatal* state.

**Definition 4.10** (Core Predicates)**.** *Let $L = (S, \Sigma, s_0, \rightarrow)$ be an IOLTS modeling an application. Let successStates be the set of success states, and let fatalStates be the set of fatal states. The model contains* critical, success, safe, and fatal state predicates *if success, fatal, critical and safe $\in P$ and the valuation function $\lambda$ has the following properties:*

$$
\begin{array}{ll}
\textit{coreAppPreds}(L, \lambda) \equiv & \\
L, \lambda, s \models \textit{success} & \textit{iff } s \in \textit{successStates} \\
L, \lambda, s \models \textit{fatal} & \in \lambda(s) \textit{ iff } s \in \textit{fatalStates} \\
L, \lambda, s \not\models \textit{success} \wedge \textit{fatal} & \\
\textit{critical} & \equiv \textbf{\textit{EX}}\,\textit{fatal} \\
\textit{safe} & \equiv \neg\, \textit{success} \wedge \neg\, \textit{critical} \wedge \neg\, \textit{fatal} \\
L, \lambda \models \textit{success}(s) & \textit{if success} \in \lambda(s) \\
L, \lambda \models \textit{fatal}(s) & \textit{if fatal} \in \lambda(s) \\
L, \lambda \models \textit{critical}(s) & \textit{if critical} \in \lambda(s) \\
L, \lambda \models \textit{safe}(s) & \textit{if safe} \in \lambda(s)
\end{array}
$$

**Definition 4.11** (Assumption Predicates)**.** *The predicates given in Definition 4.10 are mirrored by* assumption predicates *on the user's side, an assumption predicate indicates whether the user assumes that a certain property holds for the application. Let $L = (S, \Sigma, s_0, \rightarrow)$ be an IOLTS modeling a user. Let*

| | |
|---|---|
| *asmSuccessStates* | *be the set of states where the user assumes that the application is in a success state,* |
| *asmFatalStates* | *be the set of states where the user assumes that the application is in a fatal state,* |
| *asmCriticalStates* | *be the set of states where the user assumes that the application is in a critical state,* |
| *asmSafeStates* | *be the set of states where the user assumes that the application is in a safe state.* |

*A model contains* assumption predicates *if asmSuccess, asmFatal, asmCritical and asmSafe $\in P$ and*

$$
\begin{array}{ll}
\textit{coreAppPreds}(L, \lambda) \equiv & \\
L, \lambda, s \models \textit{asmSuccess} & \textit{iff } s \in \textit{asmSuccessStates} \\
L, \lambda, s \models \textit{asmFatal} & \textit{iff } s \in \textit{asmFatalStates} \\
L, \lambda, s \models \textit{asmCritical} & \textit{iff } s \in \textit{asmCriticalStates} \\
L, \lambda, s \models \textit{asmSafe} & \textit{iff } s \in \textit{asmSafe}
\end{array}
$$

## 4.3 Example

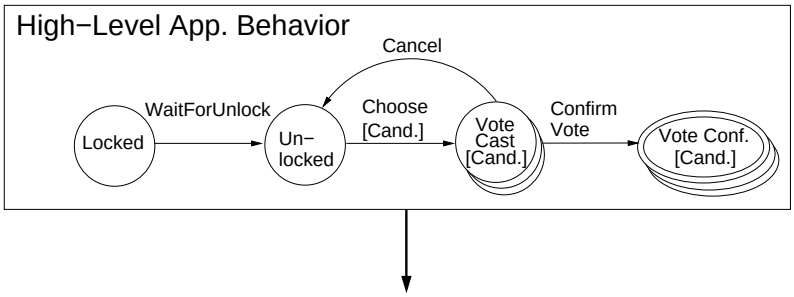


Figure 4.8: Application Model for the eVoting example.

We continue with the eVoting example from Section 4.1, in a slightly more detailed, parametrized version. We assume that the eVoting machine is initially in a locked state. After some time, the machine is unlocked and the user can cast his vote. After he has selected a candidate, the machine shows the user's choice and asks for confirmation. If he confirms, the voting process finishes. If he cancels, he can change the vote. Figure 4.8 shows an IOLTS modeling the voting machine. For $n$ candidates, "Vote Cast [Candidate]" and "Vote Confirmed [Candidate]" represent $n$ states each, one for each candidate, "Choose [Candidate]" represents the $n$ transitions to the states associated with the candidates. The input alphabet is identical to the output alphabet of the user model IOLTS, i.e., the operators available to the user. The output alphabet is an abstract representation of the application's output[2]. In this example, the sets of critical, fatal, success, and safe states are modeled as follows. Let $c$ be the candidate of choice of the user:

$$\begin{array}{ll} \textit{fatal} & \in \lambda(\text{“Vote Confirmed [}i\text{]”}) \qquad \text{for all } i \neq c \\ \textit{success} & \in \lambda(\text{“Vote Confirmed [}c\text{]”}) \end{array}$$

From the definition of *critical* it directly follows that

$$\textit{critical} \in \lambda(\text{“Vote Cast [}i\text{]”}) \quad \text{for all } i \neq c$$

and from the definition of *safe* it follows that

$$\begin{array}{l} \textit{safe}(\text{“Vote Cast [}c\text{]”}) \\ \textit{safe}(\text{“Locked”}) \\ \textit{safe}(\text{“Unlocked”}) \end{array}$$

[2]For now, we model only the top level behavior of the application. In Chapter 6 we introduce hierarchical models which allow to model all details of human-computer interaction.

# Chapter 5

# Formalized GOMS

## 5.1 Formal Semantics for GOMS User Models

We base our formalization on GOMS, because GOMS is a well established formalism, and—in the incarnation CMN-GOMS (John and Kieras, 1996)—it allows to describe user models hierarchically. This is an important property for modeling a user interface under security aspects because of the large variety of errors in human-computer interaction. Some of these errors are on a very low level (for example, the user may push the mouse button twice instead of once), while others are on a very high level of abstraction (e.g., the user may misinterpret the meaning of an error message). A hierarchical modeling mechanism allows to model all kinds of errors within one formalism. GOMS models are semi-formal. In this chapter, however, formal semantics for GOMS are defined based on the formal methods defined in Chapter 4. In Section 5.1.1, the formal semantics are extended by defining semantics of selection criteria. In combination with the formal model of the application (Section 5.2), and the formal definition of HCI security developed in the second part of this work, automated reasoning about the security of a HCI interaction model becomes possible.

GOMS describes human behavior in categories of

| | |
|---|---|
| *G*oals | The user's goals |
| *O*perators | Atomic actions available to the user |
| *M*ethods | Sequences of operators and sub-goals |
| *S*election Rules | Rules to decide between alternative methods |

In CMN-GOMS, methods for achieving a goal consist of sequences of sub-goals and atomic operators (the only difference between sub-goals and atomic operators is that operators cannot be further decomposed). If there is more than one way to reach a goal, a selection rule is used to choose between alternatives.

GOAL: VOTE FOR CANDIDATE(“Bob”)
  OPERATOR: WAIT FOR UNLOCK OF VOTING MACHINE
  OPERATOR: CHOOSE CANDIDATE(“Bob”)
  GOAL: REVIEW VOTE
  SELECT:
    OPERATOR: CONFIRM VOTE. . . if candidate “Bob” selected
    GOAL: CHANGE VOTE . . . otherwise
      OPERATOR: CANCEL VOTE
      OPERATOR: CHOOSE CANDIDATE(“Bob”)
      GOAL: REVIEW VOTE(2)
      SELECT:
        OPERATOR: CONFIRM VOTE. . . if candidate “Bob” selected
        OPERATOR: FAIL . . . otherwise

Figure 5.1: GOMS model for eVoting.

**Example** Figure 5.1 gives an example. It models the user of an eVoting machine. In order to achieve the goal “VOTE FOR CANDIDATE(‘Bob’),” the user executes the method consisting of the atomic operations “WAIT FOR UNLOCK OF VOTING MACHINE” and “CHOOSE CANDIDATE(‘Bob’)”. Then he reviews his vote. The sub-goal “REVIEW VOTE” can be achieved in two ways: (1) If the user has selected the right candidate, he confirms. (2) If he has selected the wrong candidate, he pursues sub-goal “CHANGE VOTE.” Changing the vote leads to the sub-goal “REVIEW VOTE(2).” If the user has selected the right candidate this time, he confirms; otherwise, voting fails.

**Definition** We give formal semantics for GOMS models using the notion of IOLTS traces. That is, an IOLTS corresponds to a GOMS model if the traces of the IOLTS are identical to the possible sequences of user decisions (selections) and operations. In order to formally define which IOLTS correspond to a given GOMS model, we use the following formal syntax for GOMS models:

**Definition 5.1** (Formal GOMS Model)**.** *Given a GOMS model, the corresponding* formal GOMS model *is*

$$T = (G, O, M, R, C, g_0)$$

*where*

- $G$ *is the set of (sub-)goals;*
- $O$ *is the set of operators;*

- *$C$ is the set of selection criteria;*
- *$M$ is a function mapping goals to their sequences of sub-goals/operators.*
- *A function $R : G \times C \to G$. $R$ represents the selection rules. If $R(g, c) = g'$, then goal $g$ is achieved by sub-goal/operator $g'$ in case criteria $c$ holds;*
- *$g_0$ is the top-level goal.*

The formal GOMS model corresponding to the eVoting GOMS model from Figure 5.1 is shown in Figure 5.2.

$T = (G, O, M, R, C, g_0)$ with

$$
\begin{aligned}
G &= \{\text{VOTE\_FOR\_CANDIDATE, REVIEW\_VOTE,} \\
&\quad \text{CHANGE\_VOTE, REVIEW\_VOTE(2)}\} \\
O &= \{\text{WAIT\_FOR\_UNLOCK, CHOOSE\_CANDIDATE,} \\
&\quad \text{CONFIRM\_VOTE, CANCEL\_VOTE, FAIL}\} \\
C &= \{\text{Candidate ``Bob'' selected}, \neg\,(\text{Candidate ``Bob'' selected})\} \\
g_0 &= \text{VOTE\_FOR\_CANDIDATE} \\
M(g) &= \begin{cases}
\langle \text{WAIT\_FOR\_UNLOCK}, \text{CHOOSE\_CANDIDATE}, \\
\quad \text{REVIEW\_VOTE}\rangle & \text{if } g = \text{VOTE\_FOR\_CANDIDATE} \\
\langle \text{CANCEL\_UNLOCK}, \text{CHOOSE\_CANDIDATE}, \\
\quad \text{REVIEW\_VOTE(2)}\rangle & \text{if } g = \text{CHANGE\_VOTE}
\end{cases} \\
R(g, c) &= \begin{cases}
\text{CONFIRM\_VOTE} & \text{if } g = \text{REVIEW\_VOTE and} \\
& \quad c = \text{Candidate ``Bob'' selected} \\
\text{CHANGE\_VOTE} & \text{if } g = \text{REVIEW\_VOTE and} \\
& \quad c = \neg\,(\text{Candidate ``Bob'' selected}) \\
\text{CONFIRM\_VOTE} & \text{if } g = \text{REVIEW\_VOTE(2) and} \\
& \quad c = \text{Candidate ``Bob'' selected} \\
\text{FAIL} & \text{if } g = \text{REVIEW\_VOTE(2) and} \\
& \quad c = \neg\,(\text{Candidate ``Bob'' selected})
\end{cases}
\end{aligned}
$$

Figure 5.2: Formal GOMS model for the eVoting model from Figure 5.1.

We define a formal semantics for GOMS models by translating the formal GOMS model into an IOLTS. The idea is to represent operators as elements of the output alphabet, selections as elements from the input alphabet, and methods as (sub-)paths. Selection rules are branching points in the IOLTS. Figure 5.3 illustrates this representation.

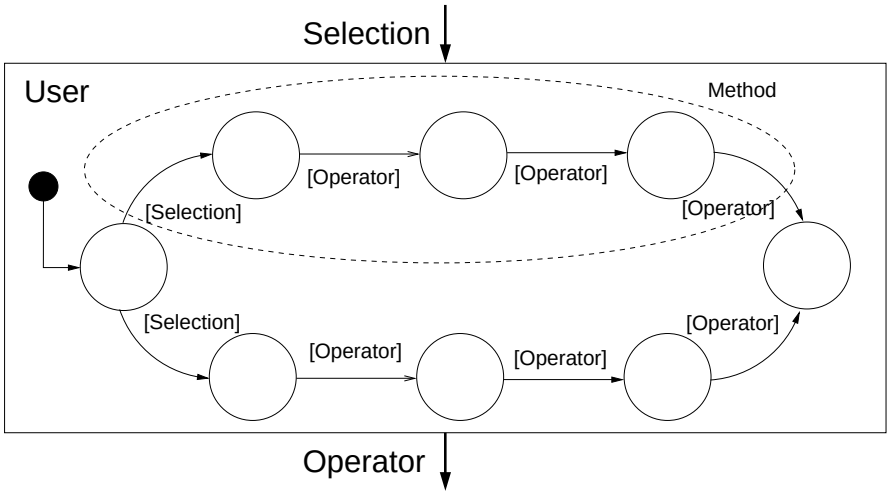


Figure 5.3: Translating GOMS categories to state transition diagrams.

**Definition 5.2** (IOLTS for GOMS Model)**.** *Let $T = (G, O, M, R, C, g_0)$ be a formal GOMS model. Let $(S, \Sigma, S_0, \rightarrow)$ be the (generalized) IOLTS constructed for $T$ by the algorithm shown in Algorithm 1. Then $(S, \Sigma, s_0, \rightarrow)$ is the* IOLTS corresponding to $T$.

**Algorithm 1** This algorithm calls the the algorithm for constructing an IOLTS corresponding to a given GOMS model (Algorithm 2) with the correct arguments.

**Require:** GOMS model $T = (G, O, M, R, c, g_0)$
**Ensure:** (Generalized) IOLTS $L = (S, \Sigma, S_0, \rightarrow)$ for $T$
1: Execute Algorithm 2 with GOMS model $T = (G, O, M, R, c, g_0)$, and $S_0 = \{s_0\}$

Algorithm alg:algo just calls Algorithm 2 with the correct arguments. Algorithm 2 creates the IOLTS recursively. The algorithm gets two inputs: a GOMS model and a set of initial states. When the algorithm is executed, the set of initial states contains $g_0$ only. The algorithm is split into three conditional parts. The part executed depends on the type of the top-level goal $g_0$. If the top-level goal is an atomic operator, lines 2 to 8 are executed. If the top-level goal is a method, lines 10 to 23 is executed. If the top-level goal is a selection rule, lines 25 to 36 are executed.

**If the goal is an atomic operator** In case the goal is in atomic operator, a new state is created, and all elements from the set of initial states are connected to the

**Algorithm 2** Algorithm for constructing an IOLTS corresponding to a given GOMS model.

**Require:** GOMS model $T = (G, O, M, R, c, g_0)$, and a set $S_0$ of initial states
**Ensure:** (Generalized) IOLTS $L = (S, \Sigma, S_0, \rightarrow)$ and set $F$ of states, s.t. $\Sigma? = C$, $\Sigma! = O$, and $F$ contains the final states of $L$

**if** $g_0 \in O$ **then**
  {initial goal is an atomic operator}
  create new state $s_1$
  $S := S_0 \cup \{s_1\}$
  $\Sigma? := \varnothing$
  $\Sigma! := \{g_0\}$
  $\rightarrow := \{(s_0, g_0, s_1) \mid s_0 \in S_0\}$
  $F := \{s_1\}$
**else if** $M(g_0) = \langle m_1, \ldots, m_n \rangle$ **then**
  {initial goal has sub-goals $m_1, \ldots, m_n$}
  $S := \varnothing$
  $\Sigma? := \varnothing$
  $\Sigma! := \varnothing$
  $\rightarrow := \varnothing$
  $F := S_0$
  **for** $i = 1 \ldots n$ **do**
    create an IOLTS $L_i = (S_i, \Sigma_i, S_0^i, \rightarrow_i)$ with final states $F_i$
      for $T_i = (G, O, M, R, c, m_i)$ and set $S_0^i := F$ of initial states
      by recursion
    $S := S \cup S_i$
    $\Sigma? := \Sigma? \cup \Sigma?_i$
    $\Sigma! := \Sigma! \cup \Sigma!_i$
    $\rightarrow := \rightarrow \cup \rightarrow_i$
    $F := F_i$
  **end for**
**else**
  {initial goal is a selection point}
  **for all** $g_i, c_i$ such that $R(g_0, c_i) = g_i$ **do**
    create a new state $s_i$
    $S := S \cup \{s_i\}$
    $\rightarrow := \rightarrow \cup \{(s_0, c_i, s_i) \mid s_0 \in S_0\}$
    create an IOLTS $L_i = (S_i, \Sigma_i, s_i, \rightarrow_i)$ with final states $F_i$
      for $T_i = (G, O, M, R, c, g_i)$ and set $S_0^i := \{s_i\}$ of initial state
      by recursion
    $S := S \cup S_i$
    $\Sigma? := \Sigma? \cup \Sigma?_i$
    $\Sigma! := \Sigma! \cup \Sigma!_i \cup \{c_i\}$
    $\rightarrow := \rightarrow \cup \rightarrow_i$
    $F := F \cup F_i$
  **end for**
**end if**

new state with label $g_0$, and $g_0$ is added to the list of output symbols. Thus, the IOLTS resulting from the transformation outputs atomic operator $g_0$.

**If the goal is a method** In case the goal is a method, the algorithm is called recursively for each element of its sequence of the sub-goals, where the final states of a sub-IOLTS are the initial states of the next sub-IOLTS.

**If the goal is a selection rule** In case the goal is a selection rule, a new state is introduced. For each of the potential sub-goals an IOLTS is constructed recursively. The newly created state is connected to the initial state of each of the sub-IOLTS, and the edge is labeled with the input symbol representing the selection criteria for the sub-IOLTS. Thus, depending on the selection criteria, the resulting IOLTS transits into the initial state of the corresponding sub-IOLTS.

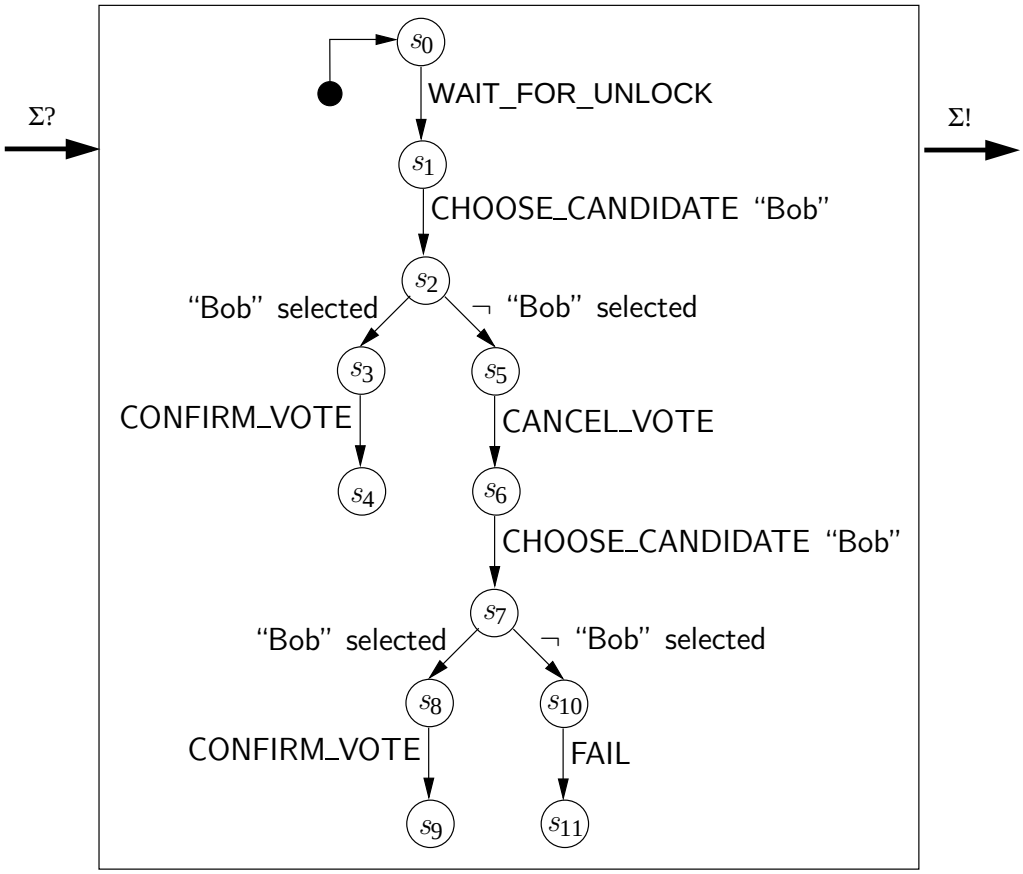


Figure 5.4: IOLTS corresponding to the eVoting GOMS model.

**Set of initial states** Algorithm 2 gets a set of initial states as an input parameter. These states are the connection points between the recursively generated sub-IOLTS. Algorithm 1 calls Algorithm 2 with the correct arguments to start recursion.

The Perl program given in Appendix B implements the algorithm. It has been used for constructing the example IOLTS presented in this thesis. Applying the algorithm to the eVoting example results in the IOLTS shown in Figure 5.4.

### 5.1.1 Assumptions as Selection Rules

Selection rules in GOMS models require decision criteria. In GOMS, these criteria are only specified in an informal way. Since our goal is to provide a formal semantics for GOMS models suitable for automated reasoning, a methodology for the formal description of selection criteria is required.

If a user is in the situation to choose between multiple options, his decision will be based on the current system configuration or, more precisely, on his *perception* of the system configuration. In the eVoting example, the decision whether to confirm his vote or to change it, depends on the candidate selection shown by the voting machine and the user's corresponding perception of the machine's internal configuration.

Following our component-based approach, we define the user's assumption about the system configuration as a component. This component is combined with the (IOLTS corresponding to the) formal GOMS model by mutual composition. The rational behind mutual composition is that not only do the user's presumptions about the application state influence his behavior but his assumptions about the state of the application are influenced by his actions as well. For example, when the user pushes the "confirm vote" button, he will assume that the voting process is completed, even if it takes some time before the next message appears on the screen. The other input for the assumption component—besides the users actions, i.e., the operators in the GOMS model—comes from the output of the application (application output is defined in Section 5.2). Figure 5.5 illustrates the composition of an interactive formal user model.

**Definition 5.3** (User Assumption IOLTS)**.** *Let* $L = (S, \Sigma, s_0, \rightarrow)$ *be an IOLTS.* $L$ *is a* user assumption IOLTS*, if*

$\Sigma = \Sigma? \cup \Sigma!$,

$\Sigma? = \Sigma?_D \cup \Sigma?_A$ *where* $\Sigma?_D$ *consists of atomic application (device) output and* $\Sigma?_A$ *contains GOMS operators,*

$\Sigma!$ *consists of GOMS selection criteria.*

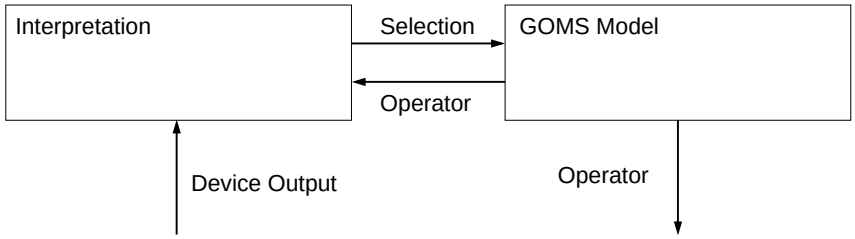


Figure 5.5: Combination of GOMS model and user's interpretation of the application's configuration.

*An* interactive formal user model $L = (L_A \parallel L_I)$ *is the mutual composition of the IOLTS* $L_U$ *corresponding to a formal GOMS model and a user assumption IOLTS* $L_I$.

### 5.1.2 Formal HCI Model: Summary

We have defined formal semantics for GOMS models and for selection criteria. Selection criteria are defined by a component modeling the user's assumptions about the application. The combination of a formal GOMS model of the user and a model of the user's assumptions allows the formal description of human behavior.

In order to reason about security of HCI, a formal application model and a formal definition of HCI security are additionally required. In Section 5.2, we complete the eVoting example with the application model from Section 4.2 and two alternative user assumption components.

## 5.2 Completing the eVoting Model

In order to apply automated reasoning to human-computer interaction, we need three components: (1) A formal GOMS model and its corresponding IOLTS; (2) a component representing the assumptions of the user about the application; and (3) a component representing the application itself. The example eVoting application has been introduced in Section 4.2. For the completion of the example, we still need a model of the user's assumptions. As defined in Section 5.1.1, a user assumption component has an input alphabet consisting of the application's output and the user's operators, and an output alphabet consisting of the user's selection criteria.

Figure 5.6 shows a user selection component for the eVoting example. In order to keep the example simple, the user assumption component takes only the application's output as input. Selection rules are used at two points in the GOMS

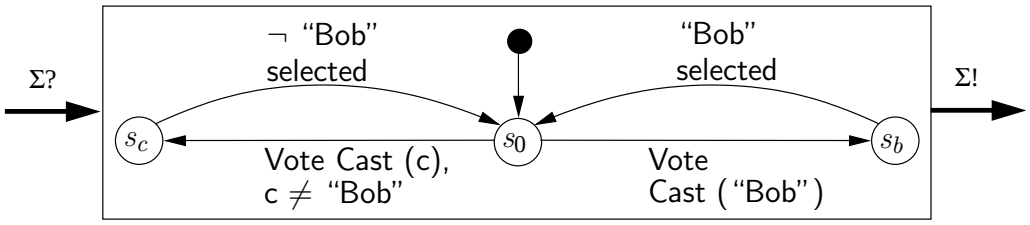


Figure 5.6: Correct user assumption component for eVoting example.

model: When the user reviews his voting decision for the first time, and when he reviews his voting decision for the second time. The user's assumption is that the eVoting application works correctly. Therefore, the assumption component will output "candidate 'Bob' selected" for the input "Vote cast('Bob')", and "¬(Candidate 'Bob' selected)" for the input "Vote cast($c$)" with $c \neq$ "Bob". This "error-free" model corresponds to the following user assumption IOLTS:

$$
\begin{aligned}
S &= \{s_0, s_b, s_c\} \\
\Sigma &= \Sigma? \cup \Sigma! \\
\Sigma? &= \{\text{locked}, \text{unlocked}\} \cup \bigcup_{c \in \text{Candidates}} \{\text{Vote cast}(c), \text{Vote confirmed}(c)\} \\
\Sigma! &= \{\text{Candidate 'Bob' selected}, \neg\,(\text{Candidate 'Bob' selected})\} \\
\rightarrow &= \{(s_0, \sigma, s_0) \mid \sigma \neq \text{Vote cast}(c) \text{ for all candidates } c\} \cup \\
&\quad \{(s_0, \text{Vote cast('Bob')}, s_b)\} \cup \\
&\quad \{(s_0, \text{Vote cast}(c), s_c) \mid c \neq \text{"Bob"}\} \cup \\
&\quad \{(s_b, \text{Candidate 'Bob' selected}, s_0)\} \cup \\
&\quad \{(s_c, \neg\,(\text{Candidate 'Bob' selected}), s_0)
\end{aligned}
$$

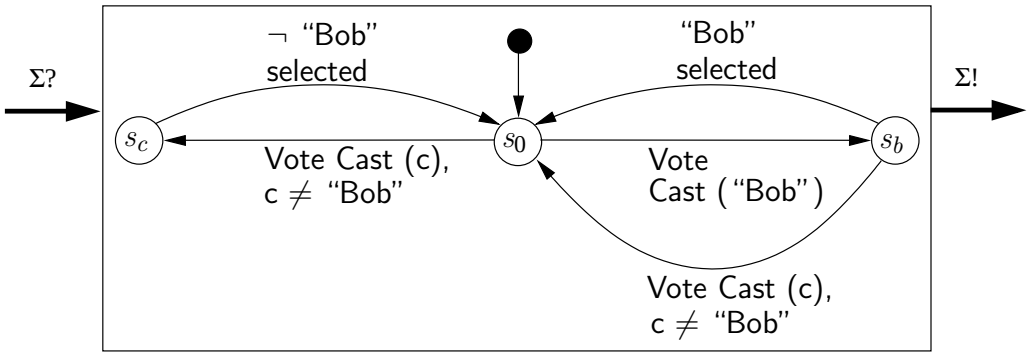


Figure 5.7: Erroneous user assumption component for eVoting example.

While standard GOMS does not allow to model user errors, our component-based approach does. As an example, we model a user who may think the system is in a state where he voted for "Bob" while in fact he voted for someone else. Figure 5.7 depicts this component. The changed relation $\rightarrow$ is shown below:

$$\begin{aligned}\rightarrow = \; & \{(s_0, \sigma, s_0) \mid \sigma \neq \text{Vote cast}(c) \text{ for all candidates } c\} \cup \\ & \{(s_0, \text{Vote cast}(c), s_b) \mid c \in \text{Candidates}\} \cup \\ & \{(s_0, \text{Vote cast}(c), s_c) \mid c \neq \text{“Bob”}\} \cup \\ & \{(s_b, \text{Candidate ‘Bob’ selected}, s_0)\} \cup \\ & \{(s_c, \neg\,(\text{Candidate ‘Bob’ selected}), s_0)\end{aligned}$$

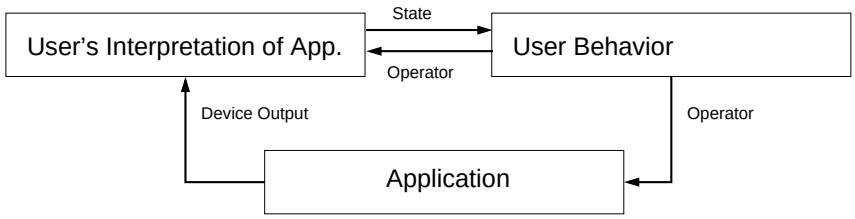


Figure 5.8: Basic system model.

In this section, we showed how system models are created from formal GOMS models, user assumption components, and application models. The mutual compositions of these three components—as shown in Figure 5.8—provide a complete model. With this complete formal modeling of human-computer interaction becomes possible. In difference to traditional methods, our method also allows to model erroneous user behavior.

In Chapter 11, we define HCI security properties as CTL formulae. Combined with a formal GOMS model of the user and a formal specification of the application, formal methods can be used for reasoning about security of human-computer interaction.

# Chapter 6

# Hierarchical Models

## 6.1 Hierarchical GOMS

In the models introduced so far, the application, the user's actions, and the user's assumptions are modeled as monolithic components. When we start to add more details to our models—for example, when application output and user perception is modeled in more detail—the components become unwieldy. We introduce hierarchical models to counter this problem. In a model of hierarchical components, components of different levels of abstraction are layered above each other. This allows to describe user interfaces and human-computer interaction at all levels of detail, while keeping each individual component small enough to be manageable by humans and computers.

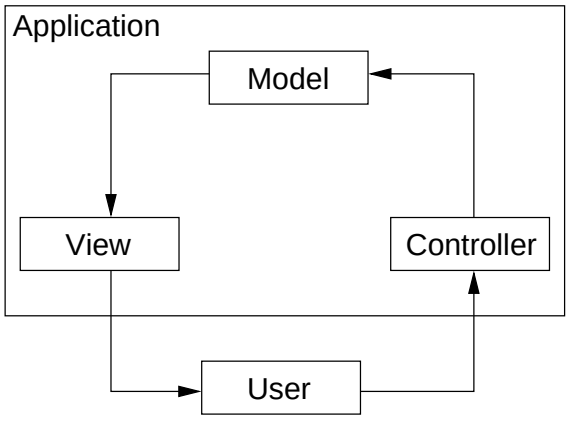


Figure 6.1: Model-View-Controller Design Pattern

Both in the construction of graphical user interfaces and in the perception (and interpretation) of graphical user interfaces, there are generic abstraction lev-

els shared over a large class of interfaces. By identifying these abstraction levels and modeling user interfaces along these lines, it becomes possible to model complex user interfaces (and potential error sources in complex user interfaces) while preserving maintainability of the models.

The seminal Model-View-Controller (MVC) design paradigm (Krasner and Pope, 1988) has been introduced using object-oriented programming for the programming language Smalltalk. MVC introduces a separation of an application model (the program logic or the data to be represented), the view on the model shown to the user, and the controller mechanisms to change the data or the state of the application, as shown in Figure 6.1.

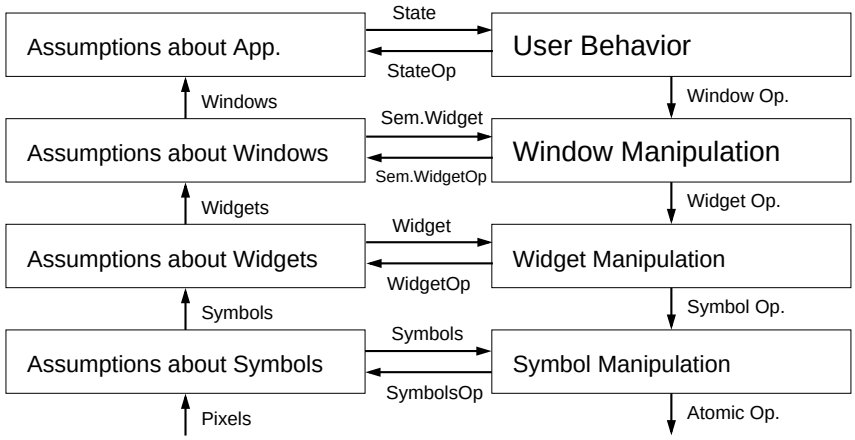


Figure 6.2: Generic Hierarchical User Model

Based on this, the controller and the viewer component can be split into subcomponents of a finer granularity. On the abstract level, a user interface allows the user to manipulate certain aspects of the program or the data. Each relevant data object is associated with a certain view object (window / screen area). Manipulation of the data object is achieved via sub-components of the screen objects (widgets) which themselves can contain widgets. On the lowest level, widgets are build from elementary symbols who themselves are constructed from pixels.

The model-view-controller pattern mirrors the distinction of formal GOMS model component and user's assumption component in the user model. On the user's side, there is a formal GOMS model governing the behavior of the user, and a model of the user's assumption's about the current state of the application. On the application's side, this is mirrored by component(s) modeling the functional behavior of the application, and component(s) visualizing the application's state. Applying the hierarchical GUI design pattern to user modeling results in a hierarchical model of human-computer interaction where each component represents one level of abstraction. This makes it possible to model typical errors on

their respective levels. For example, the typical error that a user misses the correct button and pushes a different (wrong) one instead is modeled on a low level, while the error that a user misinterprets a screen is modeled on a high level. The user's side of this design pattern is shown in Figure 6.2.

GOAL: VOTE FOR CANDIDATE("Bob")
  OPERATOR: WAIT FOR UNLOCK OF VOTING MACHINE
  OPERATOR: CHOOSE CANDIDATE("Bob")
  GOAL: REVIEW VOTE
  SELECT:
    OPERATOR: CONFIRM VOTE... if candidate "Bob" selected
    OPERATOR: FAIL ... otherwise

Figure 6.3: GOMS model for eVoting (simplified).

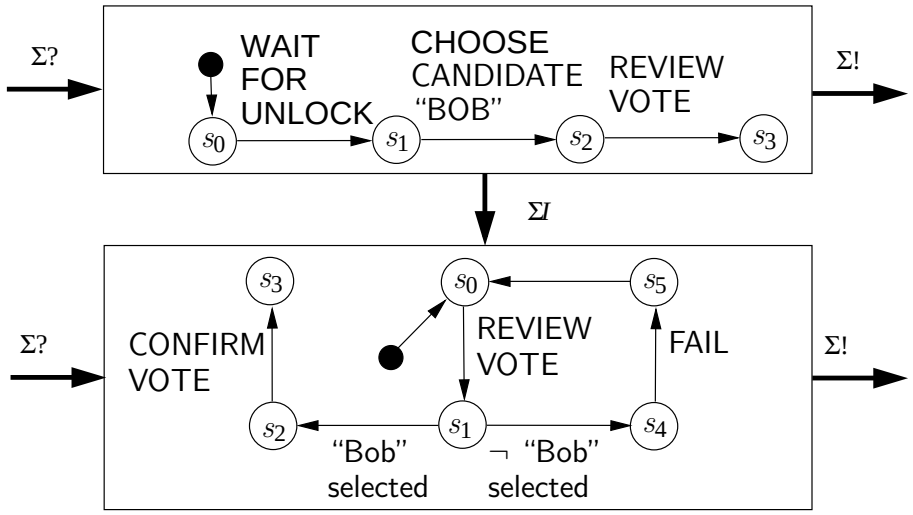


Figure 6.4: Hierarchical IOLTS corresponding to the eVoting GOMS model.

As an example, a hierarchical IOLTS for the excerpt of the eVoting GOMS model given in Figure 6.3 is shown in Figure 6.4. Note that the definition of component composition (Definition 4.3) ensures "Wait for Unlock", and "Choose Candidate 'Bob'" are part of the composed output although they are not in the alphabet of the second component. For the same reason, input "'Bob' selected", and "¬'Bob' selected" are evaluated by the second component although they are not part of the alphabet of the first component. A complete hierarchical system model consisting of user behavior components, user interpretation components, applica-

tion behavior components, and application visualization components is given in the next section.

## 6.2 Abstraction

The methodology developed in Section 6.1 allows the construction of hierarchical user and application models. A system designer can choose the level of granularity freely. The user interface can be modeled down to the pixel level, and user behavior can be modeled down to the smallest details. This makes pervasive specification and verification of human-computer interaction possible. However, in practice we face the problem of state space explosion. If we want to model the system down to the bitmap level, even a small monochrome display of size $320 \cdot 200$ pixels, as they are used e.g. in cell phones nowadays has $2^{320 \cdot 200}$ states on the lowest level. Therefore it is necessary to reduce the size of the state space. An effective method for reducing the state space is to create abstractions as defined in Definition 6.1.

**Definition 6.1** (Abstraction)**.** *Given components $A$ and $B$, a component $A$ is an abstraction of $B$ if all traces of $B$ are also traces of $A$.*

**Lemma 6.1.** *Given IOLTS $L_a = (S_a, \Sigma_a, s_{0a}, \rightarrow_a)$ and $L_b = (S_b, \Sigma_b, s_{0b}, \rightarrow_b)$, $A$ is an abstraction of $B$ if $S_a \subset S_b$, $\Sigma_a = \Sigma_b$ and there exist an abstraction function $f$ such that*

- $s_{0a} = f(s_{0b})$
- $f(s) \xrightarrow{\sigma}_a f(s')$ *if* $s \xrightarrow{\sigma}_b s'$

Abstraction is possible on all components and all hierarchies of components. It works best if the hierarchy of components in the specification of the user matches the hierarchy of components of the application. In these cases, components on the same hierarchical level can be joined together to one abstract component. This way, the lower levels of both the user behavior specification and the application specification can be successively replaced by abstract components, until the effects of human-computer interaction can be described on the top-level only.

### 6.2.1 E-Voting Example (Correct)

In Figure 6.5, the eVoting example is extended to a hierarchical model. The user and application behavior components consists of three layers of components each, and the user interpretation and the application visualization components consists of two layers each. We limit the number of hierarchy levels and the size of the

OPERATOR: WAIT FOR UNLOCK OF VOTING MACHINE
GOAL: VOTE “Bob”
  GOAL: SELECT “Bob”
  SELECT:
    OPERATOR: PUSH BUTTON 0 if “Bob” is zeroth candidate
    OPERATOR: PUSH BUTTON 1 if “Bob” is first candidate
    ...
    OPERATOR: PUSH BUTTON $n$ if “Bob” is $n$th candidate

Figure 6.5: GOMS model for hierarchical eVoting Model.

example in order to keep the model simple enough for illustration purposes, while interesting enough to demonstrate our methodology. The general strategy of the user is to vote for “Bob” once the machine is unlocked. He knows that in order to cast his vote, he has to select the right candidate from a list. When he sees the list, he looks for the entry for his candidate and pushes the corresponding button. In the example, we want to verify if the user will always cast his vote for the correct candidate, or if an error may occur, leading to a vote for a wrong candidate. We use abstraction to reduce the state space until only one state remains. If the remaining state is an abstraction of state $ab0_{\text{“Bob”}}$ and not an abstraction of $ab0_x$ for $x \neq$ “Bob”, we are guaranteed that no error can occur in the voting process.

Figure 6.6 shows the combined user and application model. On the uppermost levels (UB0 and AB0), the user and the application show the behavior that we already know from the example. The hierarchical GOMS model is given in components UB0 to UB2, and the corresponding criteria interpretation components in UI1 and UI2. The application is structured in the same way. When the voting computer is unlocked, it shows the list of candidates. In component AV2, the input symbol “Show Candidates” is expanded to the actual list of candidates, which are shown on the screen. Depending on the elements and the order of the list, the button pressed by the user is mapped to the corresponding candidate. In components, nodes with subscripts containing square brackets represent multiple states. In UB2, node $ub2_{[i]}$ stands for states $ub2_0, \ldots, ub2_{n-1}$ with $n$ the number of candidates. Transitions to and from state $ub2_{[i]}$ occur only if Bob is the $i$th candidate on the list:

$$(ub2_b, \text{Bob:i}, ub2_i) \in \rightarrow_{UB2} \quad \text{for } 0 \leq i < |\text{Candidates}|$$
$$(ub2_b, \text{Bob:j}, ub2_i) \in \rightarrow_{UB2} \quad \rightarrow i = j$$

$$(ub2_i, \text{PushButton:i}, ub2_a) \in \rightarrow_{UB2} \quad \text{for } 0 \leq i < |\text{Candidates}|$$
$$(ub2_i, \text{PushButton:j}, ub2_a) \in \rightarrow_{UB2} \quad \rightarrow i = j$$

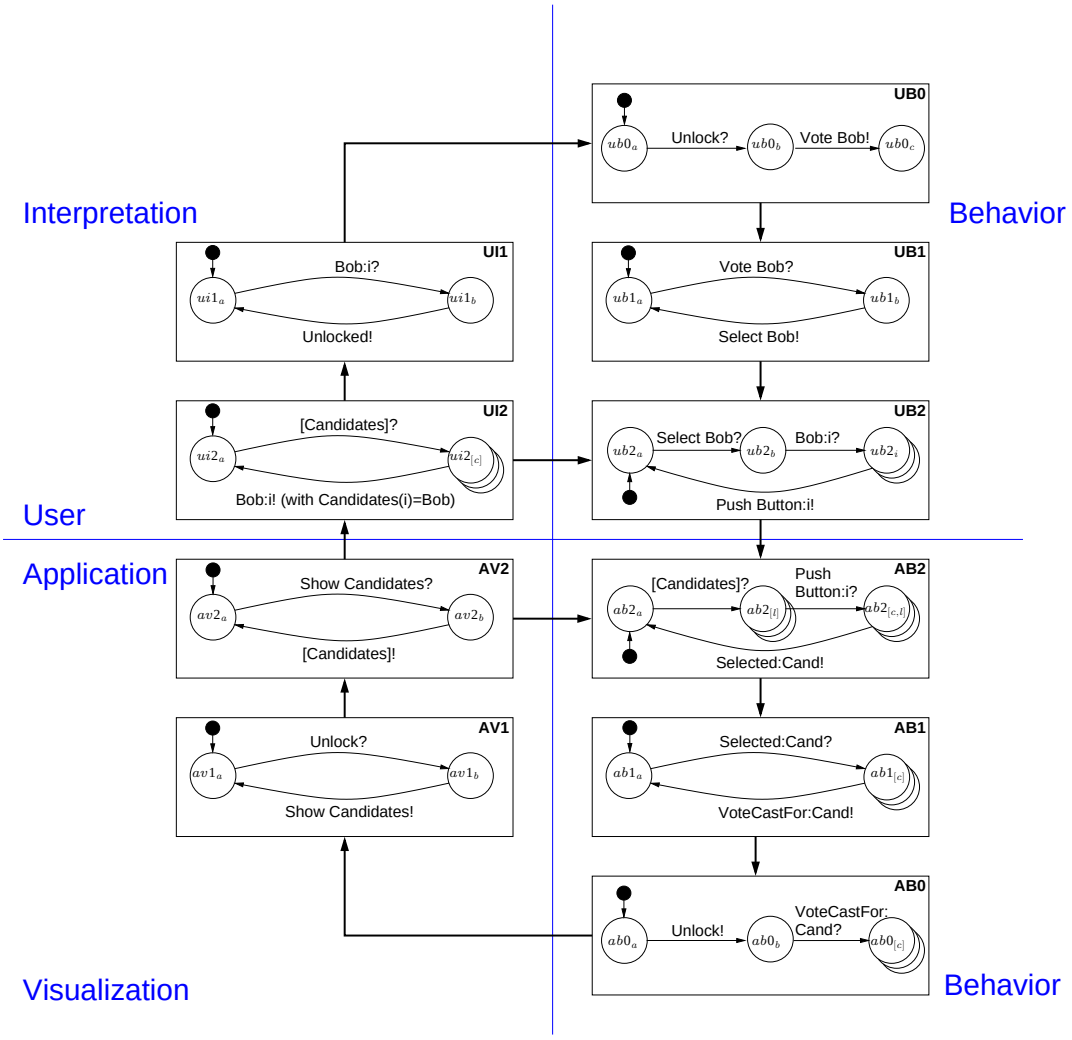


Figure 6.6: Hierarchical eVoting Model

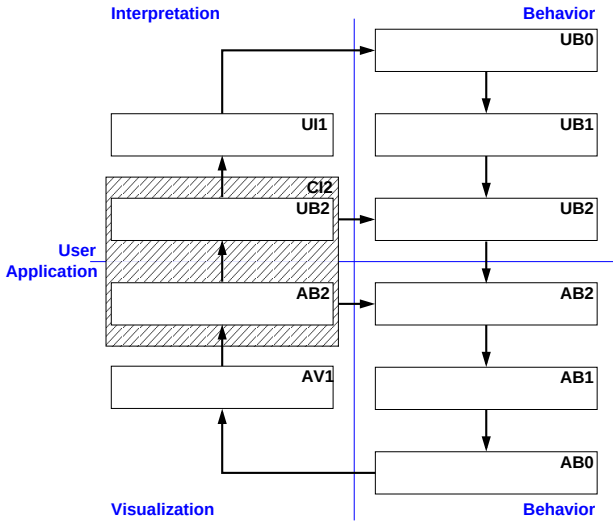


Figure 6.7: UI2 and AV2 are composed to CI2

The goal of joining components and abstraction is to reduce the state space. We start by joining the lowest level user interpretation component UI2 with the lowest level application visualization component AV2 to joined component CI2 as shown in Figure 6.7. AV2 consists of two nodes connected by two edges, while UI2 consists of $1+n$ nodes with $1+n!$ edges, given there are $n$ candidates on the list. The reason for this is the presence of one extra edge for each possible permutation of the list of candidates, and one extra node for each position of "Bob" in a list of candidates, i.e. we assume that whatever the order of the list may be, the user will identify the position of "Bob." Since the list (and thus the position of "Bob") is given by the output of AV2, joining the two components results in a component with $n-1$ less nodes and $n!-1$ less edges. AV2 is defined as

$$
\begin{aligned}
L_{av2} &= (S_{av2}, \Sigma_{av2}, s_{0\,av2}, \rightarrow_{av2}) \text{ with} \\
S_{av2} &= \{av2_a, av2_b\} \\
\Sigma_{av2} &= \Sigma?_{av2} \cup \Sigma!_{av2} \\
\Sigma?_{av2} &= \{\text{Show Candidates}\} \\
\Sigma!_{av2} &= \{[\text{Candidates}]\} \\
s_{0\,av2} &= av2_a \\
\rightarrow_{av2} &= \{(av2_a, \text{Show Candidates}, av2_b), (av2_b, [\text{Candidates}], av2_a)\}
\end{aligned}
$$

and UI2 as

$$
\begin{array}{ll}
L_{ui2} & = (S_{ui2}, \Sigma_{ui2}, s_{0\,ui2}, \rightarrow_{ui2}) \text{ with} \\
S_{ui2} & = \{ui2_a\} \cup \{ui2_i \mid 0 \leq i < |[\text{Candidates}]|\} \\
\Sigma_{ui2} & = \Sigma?_{ui2} \cup \Sigma!_{ui2} \\
\Sigma?_{ui2} & = \{\mathit{perm}(\text{Candidates})\} \\
\Sigma!_{ui2} & = \{\text{Bob} : i \mid 0 \leq i < |[\text{Candidates}]|\} \\
s_{0\,ui2} & = ui2_a \\
\rightarrow_{ui2} & = \\
\multicolumn{2}{l}{\{(ui2_a, c, av2_i) \mid \text{For all } c \in \mathit{perm}(\text{Candidates}) \text{ and } c(i) = \text{“Bob”}\}}
\end{array}
$$

The hierarchical composition $L_{CI2} = L_{av2}.L_{ui2}$ results in the following IOLTS:

$$
\begin{array}{ll}
L_{CI2} & = L_{av2}.L_{ui2} = (S, \Sigma, s_0, \rightarrow) \text{ with} \\
S & = S_{av2} \times S_{ui2} \\
\Sigma & = \Sigma? \cup \Sigma! \cup \Sigma I \\
\Sigma? & = \{\text{Show Candidates}\} \\
\Sigma! & = \{\text{Bob} : i 0 \leq i < |[\text{Candidates}]|, [\text{Candidates}]\} \\
\Sigma I & = \{\mathit{perm}(\text{Candidates})\} \\
s_0 & = (av2_a, ui2_a) \\
\rightarrow & = \\
\multicolumn{2}{l}{\{((ui2_a, av2_a), \text{Show Candidates}, (ui2_a, av2_b))\} \cup} \\
\multicolumn{2}{l}{\{((ui2_i, av2_a), \text{Show Candidates}, (ui2_i, av2_b))} \\
\multicolumn{2}{r}{\mid \text{for all } 0 \leq i < |[\text{Candidates}]|\} \cup} \\
\multicolumn{2}{l}{\{((ui2_i, av2_a), \text{Bob: } i, (ui2_a, av2_a)) \mid \text{for all } 0 \leq i < |[\text{Candidates}]|\} \cup} \\
\multicolumn{2}{l}{\{((ui2_i, av2_b), \text{Bob: } i, (ui2_a, av2_b)) \mid \text{for all } 0 \leq i < |[\text{Candidates}]|\} \cup} \\
\multicolumn{2}{l}{\{((ui2_i, av2_b), [\text{Candidates}], (ui2_i, av2_a)) \mid \text{for Candidates}[i] = \text{“Bob”}\}}
\end{array}
$$

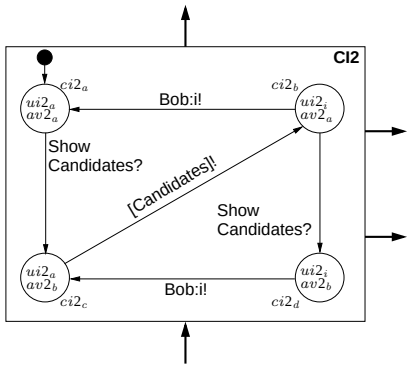


Figure 6.8: Hierarchical Composition of UI2 and AV2

From the elements of the set of nodes

$$\{(ui2_i, av2_a) \mid \text{for all } 0 \leq i < |[\text{Candidates}]|\}$$

only the node with $(ui2_j, av2_a)$ with *Candidates*$[j]$ = *"Bob"* has an incoming edge. Therefore, all other nodes from the set can be omitted. This reduced IOLTS is shown in Figure 6.8. We call this component CI2. For the following abstraction steps, we will use the shorter state names written right beside the nodes.

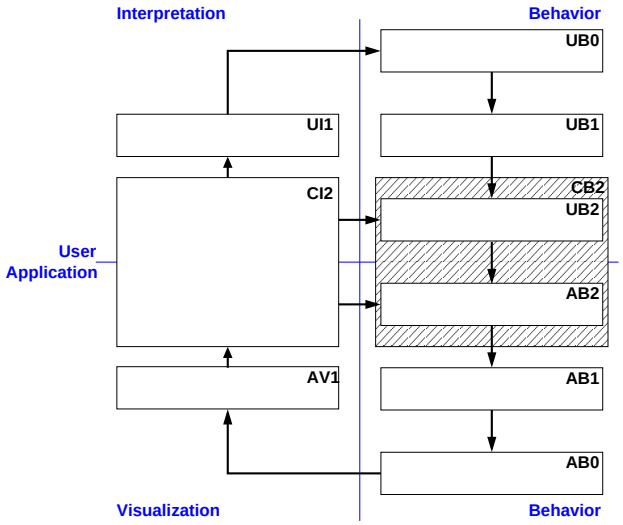


Figure 6.9: UB2 and AB2 are composed to CB2

In the next step, we compose UB2 and AB2 to CB2 as shown in Figure 6.9. UB2 is defined as:

$$\begin{aligned}
L_{ub2} &= (S_{ub2}, \Sigma_{ub2}, s_{0\,ub2}, \rightarrow_{ub2}) \text{ with} \\
S_{ub2} &= \{ub2_a, ub2_b\} \cup \{ub2_i \mid 0 \leq i < |[\text{Candidates}]|\} \\
\Sigma_{ub2} &= \Sigma?_{ub2} \cup \Sigma!_{ub2} \\
\Sigma?_{ub2} &= \{\text{Select Bob}\} \cup \{\text{Bob} : i \mid 0 \leq i < |[\text{Candidates}]|\} \\
\Sigma!_{ub2} &= \{\text{Push Button} : i \mid 0 \leq i < |[\text{Candidates}]|\} \\
s_{0\,ub2} &= ub2_a \\
\rightarrow_{ub2} &= \{(ub2_a, \text{Select Bob}, ub2_b)\} \\
&\quad \cup \{(ub2_b, \text{Bob:}i, ub2_i) \mid 0 \leq i < |[\text{Candidates}]|\} \\
&\quad \cup \{(ub2_i, \text{Push Button:}i, ub2_a) \mid 0 \leq i < |[\text{Candidates}]|\}
\end{aligned}$$

Since the only output of CI2 is *Bob:i!* for some $i$, only one of the states $ub2_0, \ldots, ub2_{n-1}$ (represented by meta-state $ub2_i$ in the component diagram of

UB2), is reachable. We can reduce UB2 to

$$
\begin{array}{ll}
L_{ub2} & = (S_{ub2}, \Sigma_{ub2}, s_{0\,ub2}, \rightarrow_{ub2}) \text{ with} \\
S_{ub2} & = \{ub2_a, ub2_b, ub2_i\} \\
\Sigma_{ub2} & = \Sigma?_{ub2} \cup \Sigma!_{ub2} \\
\Sigma?_{ub2} & = \{\text{Select Bob}, \text{Bob} : i\} \\
\Sigma!_{ub2} & = \{\text{Push Button} : i\} \\
s_{0\,ub2} & = ub2_a \\
\rightarrow_{ub2} & = \{(ub2_a, \text{Select Bob}, ub2_b), (ub2_b, \text{Bob:}i, ub2_i), \\
 & \quad (ub2_i, \text{Push Button:}i, ub2_a)\}
\end{array}
$$

We reduce the state space of AB2 in the same way. Since the order of candidates is fixed for a given eVoting setup, and UB2 will output only one specific *Push Button:i,* we can reduce AB2 to

$$
\begin{array}{ll}
L_{ab2} & = (S_{ab2}, \Sigma_{ab2}, s_{0\,ab2}, \rightarrow_{ab2}) \text{ with} \\
S_{ab2} & = \{ab2_a, ab2_l, ab2_{cl}\} \\
\Sigma_{ab2} & = \Sigma?_{ab2} \cup \Sigma!_{ab2} \\
\Sigma?_{ab2} & = \{\text{[Candidates]}, \text{Push Button:}i, \} \\
\Sigma!_{ab2} & = \{\text{Selected:“Bob”}\} \\
s_{0\,ab2} & = ab2_a \\
\rightarrow_{ab2} & = \{(ab2_a, \text{[Candidates]}, ab2_l), (ab2_l, \text{Push Button:}i, ab2_{cl}), \\
 & \quad (ab2_{cl}, \text{Selected:“Bob”}, ab2_a)\}
\end{array}
$$

Now we can compute the composition $L_{cb2} = L_{ub2}.L_{ab2}$ as

$$
\begin{array}{ll}
L_{cb2} & = (S_{cb2}, \Sigma_{cb2}, s_{0\,cb2}, \rightarrow_{cb2}) \text{ with} \\
S_{cb2} & = S_{ub2} \times S_{ab2} \\
\Sigma_{cb2} & = \Sigma?_{cb2} \cup \Sigma!_{cb2} \\
\Sigma?_{cb2} & = \{\text{Select Bob}, \text{Bob:}i, \text{[Candidates]}\} \\
\Sigma!_{cb2} & = \{\text{Selected:“Bob”}\} \\
s_{0\,cb2} & = (ub2_a, ab2a) \\
\rightarrow_{cb2} & = \\
\multicolumn{2}{l}{\{((ub2_a, ab2_a), \text{Select Bob}, (ub2_b, ab2_a)),} \\
\multicolumn{2}{l}{\ ((ub2_b, ab2_a), \text{Bob:}i, (ub2_i, ab2_a)),} \\
\multicolumn{2}{l}{\ ((ub2_a, ab2_l), \text{Select Bob}, (ub2_b, ab2_l)),} \\
\multicolumn{2}{l}{\ ((ub2_b, ab2_l), \text{Bob:}i, (ub2_i, ab2_l)),} \\
\multicolumn{2}{l}{\ ((ub2_a, ab2_{cl}), \text{Select Bob}, (ub2_b, ab2_{cl})),} \\
\multicolumn{2}{l}{\ ((ub2_b, ab2_{cl}), \text{Bob:}i, (ub2_i, ab2_{cl})),} \\
\multicolumn{2}{l}{\ ((ub2_i, ab2_l), \sigma, (ub2_a), ab2_{cl})\}} \\
\multicolumn{2}{l}{\cup\{((x, ab2_a), \text{[Candidates]}, (x, ab2_l)) \mid x \in \{ub2_a, ub2_b, ub2_i\}\}} \\
\multicolumn{2}{l}{\cup\{((x, ab2_{cl}), \text{Selected:“Bob”}, (x, ab2_a)) \mid x \in \{ub2_a, ub2_b, ub2_i\}\}}
\end{array}
$$

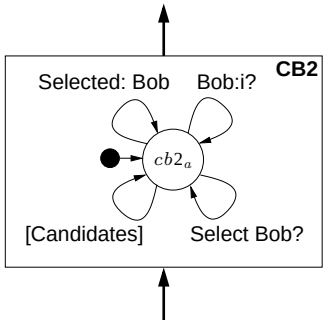


Figure 6.10: Component CB2', an abstraction of CB2

Note that *Push Button:i* is not in the output alphabet. We were able to replace it by an $\varepsilon$-transition, because UB2 has no outgoing connections. The number of nodes and edges in $L_{cb2}$ is becoming unwieldy, therefore we abstract $L_{cb2}$ to a simpler component $L_{cb2'}$. The behavior of $L_{cb2}$ can be roughly described as follows: It takes *Select Bob*, *Candidates*, and *Bob:i* as inputs and produces the output *Selected:"Bob"*. We replace $L_{cb2}$ by the simpler IOLTS $L_{cb2'}$ and prove that $L_{cb2'}$ as shown in Figure 6.10 is an abstraction of $L_{cb2}$.

$$
\begin{array}{ll}
L_{cb2'} & = (S_{cb2'}, \Sigma_{cb2'}, s_{0\,cb2'}, \rightarrow_{cb2'}) \text{ with} \\
S_{cb2'} & = \{cb2_a\} \\
\Sigma_{cb2'} & = \Sigma?_{cb2'} \cup \Sigma!_{cb2'} \\
\Sigma?_{cb2'} & = \{\text{Select Bob}, \text{Bob:}i, \text{[Candidates]}\} \\
\Sigma!_{cb2'} & = \{\text{Selected:"Bob"}\} \\
s_{0\,cb2'} & = cb2_a \\
\rightarrow_{cb2'} & = \{(cb2_a, \text{Select Bob}, cb2_a), (cb2_a, \text{Bob:}i, cb2_a), \\
 & \quad (cb2_a, \text{[Candidates]}, cb2_a), (cb2_a, \text{Selected:"Bob"}, cb2_a)\}
\end{array}
$$

**Proof**

To show: $L_{cb2'}$ is an abstraction of $L_{cb2}$.
Let abstraction function $f : S_{cb2} \rightarrow S_{cb2'}$ be:

$f(s) = cb2_a$ for all $s \in S_{cb2}$

- $s_{0\,cb2'} = f(s_{0\,cb2}) \Longleftrightarrow cb2_a = cb2_a$
- $f(s) \xrightarrow{\sigma}_a f(s')$ if $s \xrightarrow{\sigma}_b s'$

  Since $f(s) = cb2_a$ for all $s$ and $cb2_a \xrightarrow{\sigma} cb2_a$ for all $\sigma \in \Sigma_{cb2}$, $f(s) \xrightarrow{\sigma}_a f(s')$ if $s \xrightarrow{\sigma}_b s'$ holds. □

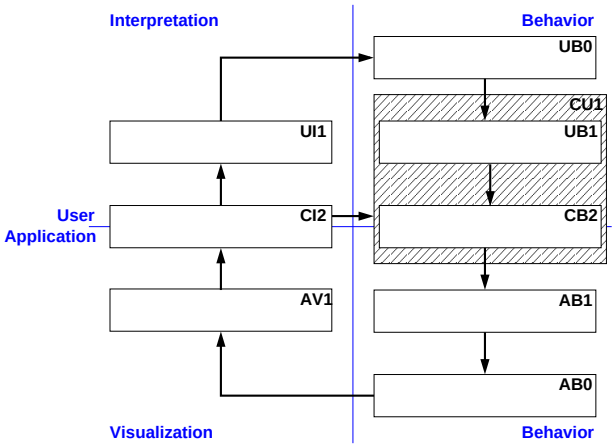


Figure 6.11: UB1 and CB2 are composed to CU1

We continue by computing the composition of $L_{cu1} = L_{ub1}.L_{cb2}$ (shown in Figure 6.11) as

$$
\begin{aligned}
L_{cu1} &= (S_{cu1}, \Sigma_{cu1}, s_{0\,cu1}, \rightarrow_{cu1}) \text{ with} \\
S_{cu1} &= S_{ub1} \times S_{cb2} \\
\Sigma_{cu1} &= \Sigma?_{cu1} \cup \Sigma!_{cu1} \\
\Sigma?_{cu1} &= \{\text{Vote Bob}\} \\
\Sigma!_{cu1} &= \{\text{Select Bob}, \text{Bob:}i, [\text{Candidates}], \text{Selected:“Bob”}\} \\
s_{0\,cu1} &= (ub1_a, cb2a) \\
\rightarrow_{cu1} &= \{((ub1_a, cb2_a), \text{Vote Bob}, (ub1_b, cb2_a)), \\
&\quad ((ub1_a, cb2_a), \text{Selected Bob}, (ub1_a, cb2_a)), \\
&\quad ((ub1_a, cb2_a), \text{Bob:}i, (ub1_a, cb2_a)), \\
&\quad ((ub1_a, cb2_a), [\text{Candidates}], (ub1_a, cb2_a)), \\
&\quad ((ub1_b, cb2_a), \varepsilon, (ub1_a, cb2_a)), \\
&\quad ((ub1_b, cb2_a), \text{Selected Bob}, (ub1_b, cb2_a)), \\
&\quad ((ub1_b, cb2_a), \text{Bob:}i, (ub1_b, cb2_a)), \\
&\quad ((ub1_b, cb2_a), [\text{Candidates}], (ub1_b, cb2_a))\}
\end{aligned}
$$

Joining the states connected by the $\varepsilon$-transition and renaming the name of the

remaining state, we get

$$
\begin{aligned}
L_{cu1} &= (S_{cu1}, \Sigma_{cu1}, s_{0\,cu1}, \rightarrow_{cu1}) \text{ with} \\
S_{cu1} &= cu1_a \\
\Sigma_{cu1} &= \Sigma?_{cu1} \cup \Sigma!_{cu1} \\
\Sigma?_{cu1} &= \{\text{Vote Bob}\} \\
\Sigma!_{cu1} &= \{\text{Select Bob}, \text{Bob:}i, [\text{Candidates}], \text{Selected:“Bob”}\} \\
s_{0\,cu1} &= cu1_a \\
\rightarrow_{cu1} &= \{(cu1_a, \text{Vote Bob}, cu1_a), \\
&\quad (cu1_a, \text{Selected: “Bob”}, cu1_a), \\
&\quad (cu1_a, \text{Bob:}i, cu1_a), \\
&\quad (cu1_a, [\text{Candidates}], cu1_a)\}
\end{aligned}
$$

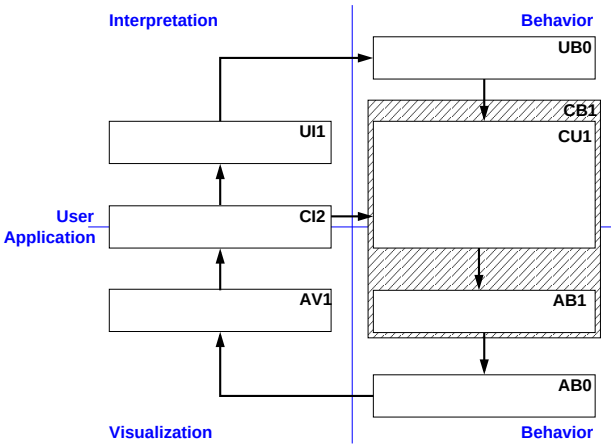


Figure 6.12: CU1 and AB1 are composed to CB1

Computing CB1 as the composition of CU1 and AB1 (see Figure 6.12) results in (the $\varepsilon$-transition has already been eliminated and the state name renamed):

$$
\begin{aligned}
L_{cb1} &= (S_{cb1}, \Sigma_{cb1}, s_{0\,cb1}, \rightarrow_{cb1}) \text{ with} \\
S_{cb1} &= cb1_a \\
\Sigma_{cb1} &= \Sigma?_{cb1} \cup \Sigma!_{cb1} \\
\Sigma?_{cb1} &= \{\text{Vote Bob}\} \\
\Sigma!_{cb1} &= \{\text{Bob:}i, [\text{Candidates}], \text{VoteCastFor:“Bob”}\} \\
s_{0\,cb1} &= cb1_a \\
\rightarrow_{cb1} &= \{(cb1_a, \text{Vote Bob}, cb1_a), (cb1_a, \text{VoteCastFor:“Bob”}, cb1_a), \\
&\quad (cb1_a, \text{Bob:}i, cb1_a), (cb1_a, [\text{Candidates}], cb1_a)\}
\end{aligned}
$$

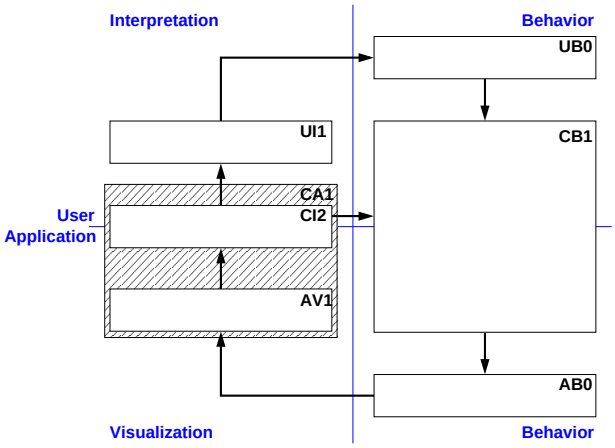


Figure 6.13: AV1 and CI2 are composed to CA1

In the same way, AV1 and CI2 are composed to CA1 (see Figure 6.13:

$$
\begin{aligned}
L_{ca1} &= (S_{ca1}, \Sigma_{ca1}, s_{0\,ca1}, \rightarrow_{ca1}) \text{ with} \\
S_{ca1} &= S_{av1} \times S_{ci2} \\
\Sigma_{ca1} &= \Sigma?_{ca1} \cup \Sigma!_{ca1} \\
\Sigma?_{ca1} &= \{\text{Unlock}\} \\
\Sigma!_{ca1} &= \{\text{Bob:}i, [\text{Candidates}]\} \\
s_{0\,ca1} &= (av1_a, ci2a) \\
\rightarrow_{ca1} &= \{(av1_a, x), \text{Unlock}, (av1_b, x) \mid x \in \{ci2_a, ci2_b, ci2_c, ci2_d\}\} \\
&\quad \cup \{((av1_a, ci2_d), \text{Bob:}i, (av1_a, ci2_c)), \\
&\qquad ((av1_a, ci2_c), [\text{Candidates}], (av1_a, ci2_b)), \\
&\qquad ((av1_a, ci2_b), \text{Bob:}i, (av1_a, ci2_a)), \\
&\qquad ((av1_b, ci2_d), \text{Bob:}i, (av1_b, ci2_c)), \\
&\qquad ((av1_b, ci2_c), [\text{Candidates}], (av1_b, ci2_b)), \\
&\qquad ((av1_b, ci2_b), \text{Bob:}i, (av1_b, ci2_a)), \\
&\qquad ((av1_b, ci2_a), \varepsilon, (av1_a, ci2_c)), \\
&\qquad ((av1_b, ci2_b), \varepsilon, (av1_a, ci2_d))\}
\end{aligned}
$$

Again, we create an abstraction and prove correctness of it:

$$
\begin{array}{ll}
L_{ca1'} & = (S_{ca1'}, \Sigma_{ca1'}, s_{0\,ca1'}, \rightarrow_{ca1'}) \text{ with} \\
S_{ca1'} & = \{ca1_a\} \\
\Sigma_{ca1'} & = \Sigma?_{ca1'} \cup \Sigma!_{ca1'} \\
\Sigma?_{ca1'} & = \{\text{Unlock}\} \\
\Sigma!_{ca1'} & = \{\text{Bob:}i, [\text{Candidates}]\} \\
s_{0\,ca1'} & = ca1_a \\
\rightarrow_{ca1'} & = \{(ca1_a, \text{Unlock}, ca1_a), (ca1_a, \text{Bob:}i, ca1_a), \\
 & \quad (ca1_a, [\text{Candidates}], ca1_a)\}
\end{array}
$$

**Proof**

To show: $L_{ca1'}$ is an abstraction of $L_{ca1}$.
Let abstraction function $f : S_{ca1} \rightarrow S_{ca1'}$ be:

$f(s) = ca1_a$ for all $s \in S_{ca1}$

- $s_{0\,ca1'} = f(s_{0\,ca1}) \Longleftrightarrow ca1_a = ca1_a$
- $f(s) \xrightarrow{\sigma}_a f(s')$ if $s \xrightarrow{\sigma}_b s'$

  Since $f(s) = ca1_a$ for all $s$ and $ca1_a \xrightarrow{\sigma} ca1_a$ for all $\sigma \in \Sigma_{ca1}$, $f(s) \xrightarrow{\sigma}_a f(s')$ if $s \xrightarrow{\sigma}_b s'$ holds. □

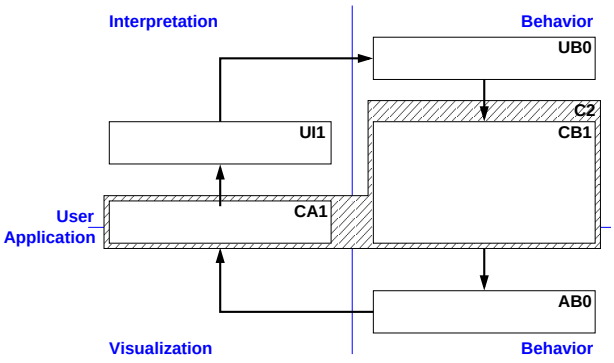


Figure 6.14: CA1 and CB1 are composed to C2

Next, we compose CA1' and CB1 to C2 (see Figure 6.14):

$$
\begin{aligned}
L_{c2} &= (S_{c2}, \Sigma_{c2}, s_{0\,c2}, \rightarrow_{c2}) \text{ with} \\
S_{c2} &= \{c2_a\} \\
\Sigma_{c2} &= \Sigma?_{c2} \cup \Sigma!_{c2} \\
\Sigma?_{c2} &= \{\text{Unlock}, \text{Vote Bob}\} \\
\Sigma!_{c2} &= \{\text{Bob:}i\text{Vote Cast For: ``Bob''}\} \\
s_{0\,c2} &= c2_a \\
\rightarrow_{c2} &= \{(c2_a, \text{Unlock}, c2_a), (c2_a, \text{Bob:}i, c2_a), \\
&\quad (c2_a, \text{Vote Cast For: ``Bob''}, c2_a), \\
&\quad (c2_a, \text{Vote Bob}, c2_a)\}
\end{aligned}
$$

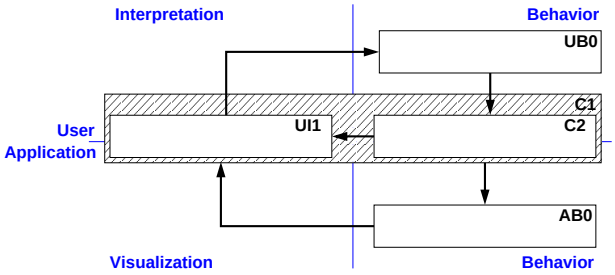


Figure 6.15: C2 and UI1 are composed to C1

C2 and UI1 are composed to C1 (see Figure 6.15):

$$
\begin{aligned}
L_{c1} &= (S_{c1}, \Sigma_{c1}, s_{0\,c1}, \rightarrow_{c1}) \text{ with} \\
S_{c1} &= \{ca1_a\} \\
\Sigma_{c1} &= \Sigma?_{c1} \cup \Sigma!_{c1} \\
\Sigma?_{c1} &= \{\text{Unlock}, \text{Vote Bob}\} \\
\Sigma!_{c1} &= \{\text{ReadyVote Cast For: ``Bob''}\} \\
s_{0\,c1} &= ca1_a \\
\rightarrow_{c1} &= \{(c1_a, \text{Unlock}, c1_a), (c1_a, \text{Ready}, c1_a), \\
&\quad (c1_a, \text{Vote Cast For: ``Bob''}, c1_a), (c1_a, \text{Vote Bob}, c1_a)\}
\end{aligned}
$$

UB0 and C1 are composed to CU0 (see Figure 6.16):

$$
\begin{aligned}
L_{c1} &= (S_{c1}, \Sigma_{c1}, s_{0\,c1}, \rightarrow_{c1}) \text{ with} \\
S_{c1} &= \{c1_a\} \\
\Sigma_{c1} &= \Sigma?_{c1} \cup \Sigma!_{c1} \\
\Sigma?_{c1} &= \{\text{Unlock}\} \\
\Sigma!_{c1} &= \{\text{Vote Cast For: ``Bob''}\} \\
s_{0\,c1} &= c1_a \\
\rightarrow_{c1} &= \{(c1_a, \text{Unlock}, c1_a), (c1_a, \text{Vote Cast For: ``Bob''}, c1_a)\}
\end{aligned}
$$

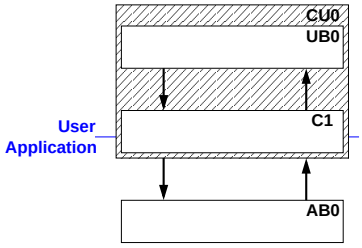


Figure 6.16: UB0 and C1 are composed to CU0

In the last step, CU0 and AB0 are composed. Since the output alphabet of CU0 is restricted to {Vote Cast For: "Bob"}, all $ab0_c$ for $c \neq$ "Bob" are unreachable. Therefore, we can reduce AB0 to

$$
\begin{array}{ll}
L_{ab0} & = (S_{ab0}, \Sigma_{ab0}, s_{0\,ab0}, \rightarrow_{ab0}) \text{ with} \\
S_{ab0} & = \{ab0_a, ab0_b, ab0_c\} \\
\Sigma_{ab0} & = \Sigma?_{ab0} \cup \Sigma!_{ab0} \\
\Sigma?_{ab0} & = \{\text{Vote Cast For: "Bob"}\} \\
\Sigma!_{ab0} & = \{\text{Unlock}\} \\
s_{0\,ab0} & = ab0_a \\
\rightarrow_{ab0} & = \{(ab0_a, \text{Unlock}, ab0_b), (ab0_b, \text{Vote Cast For: "Bob"}, ab0_c)\}
\end{array}
$$

Finally, C as the composition of CU0 and AB0 is computed as

$$
\begin{array}{ll}
L_c & = (S_c, \Sigma_c, s_{0\,c}, \rightarrow_c) \text{ with} \\
S_c & = \{c_a\} \\
\Sigma_c & = \Sigma?_c \cup \Sigma!_c \\
\Sigma?_c & = \{\} \\
\Sigma!_c & = \{\} \\
s_{0\,c} & = c_a \\
\rightarrow_c & = \{\}
\end{array}
$$

In Section 4.2 we defined

$$
\begin{array}{lll}
\textit{fatal} & \in \lambda(\text{"Vote Confirmed [}i\text{]"}) & \text{for all } i \neq c \\
\textit{success} & \in \lambda(\text{"Vote Confirmed [}c\text{]"}) &
\end{array}
$$

State $c_a$ is a result of applying the composition rule to state $ab_{\text{"Bob"}}$ and a number of other states, but non of the other state was the final voting state for a candidate other than "Bob." Therefore, we have shown that in the given model, a *success* final state is always reached, and a *fatal* final state is never reached. We have shown that the model represents the intended functionality.

### 6.2.2 E-Voting Example (with erroneous user behavior)

Next, we apply our methodology to a model with erroneous user behavior. In this example, we model a user with a shaky hand. He may accidently push the button directly above or below the intended button. This potential error is modeled on the symbol manipulation level by the GOMS sub-model for goal "Select Bob" (UB2). A changed GOMS model for this sub-goal is shown in Figure 6.17. Here, the selection of the right or wrong button is non-deterministic.

GOAL: SELECT "Bob"
SELECT:
- OPERATOR: PUSH BUTTON 0 if "Bob" is zeroth candidate
- OPERATOR: PUSH BUTTON 1 if "Bob" is zeroth candidate
- OPERATOR: PUSH BUTTON 0 if "Bob" is first candidate
- OPERATOR: PUSH BUTTON 1 if "Bob" is first candidate
- OPERATOR: PUSH BUTTON 2 if "Bob" is first candidate
- ...
- OPERATOR: PUSH BUTTON $n-1$ if "Bob" is $n$th candidate
- OPERATOR: PUSH BUTTON $n$ if "Bob" is $n$th candidate

Figure 6.17: GOMS model for hierarchical eVoting model with erroneous user behavior

With this definition, UB2 is defined as

$$
\begin{aligned}
L_{ub2} &= (S_{ub2}, \Sigma_{ub2}, s_{0\,ub2}, \rightarrow_{ub2}) \text{ with} \\
S_{ub2} &= \{ub2_a, ub2_b\} \cup \{ub2_i \mid 0 \leq i < |[\text{Candidates}]|\} \\
\Sigma_{ub2} &= \Sigma?_{ub2} \cup \Sigma!_{ub2} \\
\Sigma?_{ub2} &= \{\text{Select Bob}\} \cup \{\text{Bob} : i \mid 0 \leq i < |[\text{Candidates}]|\} \\
\Sigma!_{ub2} &= \{\text{Push Button} : i \mid 0 \leq i < |[\text{Candidates}]|\} \\
s_{0\,ub2} &= ub2_a \\
\rightarrow_{ub2} &= \{(ub2_a, \text{Select Bob}, ub2_b)\} \\
&\quad \cup \{(ub2_b, \text{Bob:}i, ub2_i) \mid 0 \leq i < |[\text{Candidates}]|\} \\
&\quad \cup \{(ub2_b, \text{Bob:}i, ub2_{i+1}) \mid 0 \leq i < |[\text{Candidates}]| - 1\} \\
&\quad \cup \{(ub2_b, \text{Bob:}i, ub2_{i-1}) \mid 0 < i < |[\text{Candidates}]|\} \\
&\quad \cup \{(ub2_i, \text{Push Button:}i, ub2_a) \mid 0 \leq i < |[\text{Candidates}]|\}
\end{aligned}
$$

Without loss of generality, we assume that Bob is neither the first nor the last candidate on the list. Since the only output of CI2 is *Bob:i!* for some $i$, only three of the states $ub2_0, \ldots, ub2_{n-1}$ (represented by meta-state $ub2_i$ in the component

diagram of UB2) are reachable. We can reduce UB2 to

$$
\begin{array}{ll}
L_{ub2} & = (S_{ub2}, \Sigma_{ub2}, s_{0\,ub2}, \rightarrow_{ub2}) \text{ with} \\
S_{ub2} & = \{ub2_a, ub2_b, ub2_i\} \\
\Sigma_{ub2} & = \Sigma?_{ub2} \cup \Sigma!_{ub2} \\
\Sigma?_{ub2} & = \{\text{Select Bob}, \text{Bob} : i\} \\
\Sigma!_{ub2} & = \{\text{Push Button} : i-1, \text{Push Button} : i, \text{Push Button} : i+1\} \\
s_{0\,ub2} & = ub2_a \\
\rightarrow_{ub2} & = \{(ub2_a, \text{Select Bob}, ub2_b), (ub2_b, \text{Bob:}i, ub2_i), \\
 & \quad (ub2_i, \text{Push Button:}i-1, ub2_a), (ub2_i, \text{Push Button:}i, ub2_a), \\
 & \quad (ub2_i, \text{Push Button:}i+1, ub2_a)\}
\end{array}
$$

We apply the same technique to AB2. Since the order of the candidates is fixed for a given eVoting setup, and UB2 will possibly output only *Push Button:i-1*,*Push Button:i*, or *Push Button:i+1*, we can reduce the composed model to these three possibilities. Without loss of generality, we assume the candidate in the list above "Bob" is "Alice" and the candidate below "Bob" is "Charlie." We can reduce AB2 to

$$
\begin{array}{ll}
L_{ab2} & = (S_{ab2}, \Sigma_{ab2}, s_{0\,ab2}, \rightarrow_{ab2}) \text{ with} \\
S_{ab2} & = \{ab2_a, ab2_l, ab2_c l\} \\
\Sigma_{ab2} & = \Sigma?_{ab2} \cup \Sigma!_{ab2} \\
\Sigma?_{ab2} & = \{\text{[Candidates]}, \text{Push Button:}i, \text{Push Button:}i-1, \\
 & \quad \text{Push Button:}i+1, \} \\
\Sigma!_{ab2} & = \{\text{Selected:“Alice”}, \text{Selected:“Bob”}, \text{Selected:“Charlie”}\} \\
s_{0\,ab2} & = ab2_a \\
\rightarrow_{ab2} & = \{(ab2_a, \text{[Candidates]}, ab2_l), \\
 & \quad (ab2_l, \text{Push Button:}i-1, ab2_{cl'}), \\
 & \quad (ab2_l, \text{Push Button:}i, ab2_{cl}), \\
 & \quad (ab2_l, \text{Push Button:}i+1, ab2_{cl''}), \\
 & \quad (ab2_{cl'}, \text{Selected::“Alice”}, ab2_a), \\
 & \quad (ab2_{cl}, \text{Selected::“Bob”}, ab2_a), \\
 & \quad (ab2_{cl''}, \text{Selected::“Charlie”}, ab2_a)\}
\end{array}
$$

We construct $L_{cb2} = L_{ub2}.L_{ab2}$ as in Section 6.2.1 (see Figure 6.9) and apply

the same abstraction step, resulting in

$$
\begin{array}{ll}
L_{cu1} & = (S_{cu1}, \Sigma_{cu1}, s_{0\,cu1}, \rightarrow_{cu1}) \text{ with} \\
S_{cu1} & = \{cu1_a\} \\
\Sigma_{cu1} & = \Sigma?_{cu1} \cup \Sigma!_{cu1} \\
\Sigma?_{cu1} & = \{\text{Vote Bob}\} \\
\Sigma!_{cu1} & = \{\text{Select Bob}, \text{Bob:}i, \text{[Candidates]}, \text{Selected:“Alice”}, \\
 & \quad \text{Selected:“Bob”}, \text{Selected:“Charlie”}\} \\
s_{0\,cu1} & = cu1_a \\
\rightarrow_{cu1} & = \{(cu1_a, \text{Vote Bob}, cu1_a) \\
 & \quad (cu1_a, \text{Selected: “Alice”}, cu1_a), \\
 & \quad (cu1_a, \text{Selected: “Bob”}, cu1_a), \\
 & \quad (cu1_a, \text{Selected: “Charlie”}, cu1_a), \\
 & \quad (cu1_a, \text{Bob:}i, cu1_a), \\
 & \quad (cu1_a, \text{[Candidates]}, cu1_a)\}
\end{array}
$$

We continue by computing the composition of $L_{cu1} = L_{ub1}.L_{cb2}$ (see Figure 6.11). Joining the states connected by the $\varepsilon$-transition and renaming the remaining state, we get

$$
\begin{array}{ll}
L_{cu1} & = (S_{cu1}, \Sigma_{cu1}, s_{0\,cu1}, \rightarrow_{cu1}) \text{ with} \\
S_{cu1} & = \{cu1_a\} \\
\Sigma_{cu1} & = \Sigma?_{cu1} \cup \Sigma!_{cu1} \\
\Sigma?_{cu1} & = \{\text{Vote Bob}\} \\
\Sigma!_{cu1} & = \{\text{Select Bob}, \text{Bob:}i, \text{[Candidates]}, \text{Selected:“Alice”}, \\
 & \quad \text{Selected:“Bob”}, \text{Selected:“Charlie”}\} \\
s_{0\,cu1} & = cu1_a \\
\rightarrow_{cu1} & = \{(cu1_a, \text{Vote Bob}, cu1_a), (cu1_a, \text{Selected: “Alice”}, cu1_a), \\
 & \quad (cu1_a, \text{Selected: “Bob”}, cu1_a), \\
 & \quad (cu1_a, \text{Selected: “Charlie”}, cu1_a), (cu1_a, \text{Bob:}i, cu1_a), \\
 & \quad (cu1_a, \text{[Candidates]}, cu1_a)\}
\end{array}
$$

Computing CB1 as the composition of CU1 and AB1 (see Figure 6.12) results in (the $\varepsilon$-transition has already been eliminated and the state name renamed) the

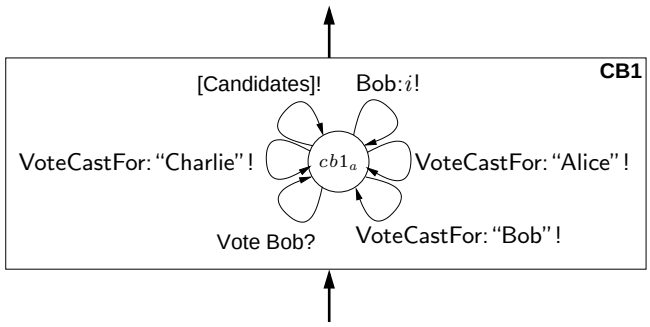


Figure 6.18: Component CB1

IOLTS shown in Figure 6.18:

$$
\begin{array}{ll}
L_{cb1} & = (S_{cb1}, \Sigma_{cb1}, s_{0\,cb1}, \rightarrow_{cb1}) \text{ with} \\
S_{cb1} & = cb1_a \\
\Sigma_{cb1} & = \Sigma?_{cb1} \cup \Sigma!_{cb1} \\
\Sigma?_{cb1} & = \{\text{Vote Bob}\} \\
\Sigma!_{cb1} & = \{\text{Bob:}i, \text{[Candidates]}, \text{VoteCastFor:“Alice”}, \\
 & \quad \text{VoteCastFor:“Bob”}, \text{VoteCastFor:“Charlie”}\} \\
s_{0\,cb1} & = cb1_a \\
\rightarrow_{cb1} & = \{(cb1_a, \text{Vote Bob}, cb1_a) \\
 & \quad (cb1_a, \text{VoteCastFor:“Alice”}, cb1_a), \\
 & \quad (cb1_a, \text{VoteCastFor:“Bob”}, cb1_a), \\
 & \quad (cb1_a, \text{VoteCastFor:“Charlie”}, cb1_a), \\
 & \quad (cb1_a, \text{Bob:}i, cb1_a), \\
 & \quad (cb1_a, \text{[Candidates]}, cb1_a)\}
\end{array}
$$

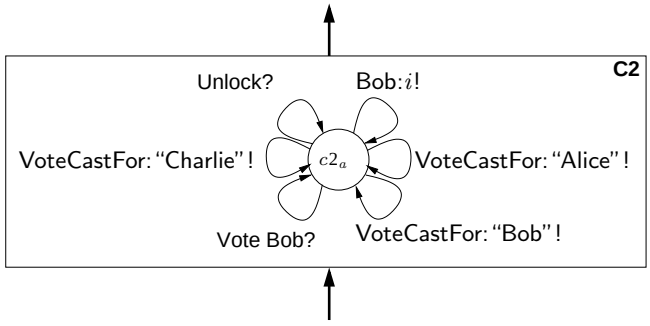


Figure 6.19: Component C2

The composition of CA1' and CB1 to C2 (see Figure 6.14) leads to the IOLTS shown in Figure 6.19:

$$
\begin{aligned}
L_{c2} &= (S_{c2}, \Sigma_{c2}, s_{0\,c2}, \rightarrow_{c2}) \text{ with} \\
S_{c2} &= \{c2_a\} \\
\Sigma_{c2} &= \Sigma?_{c2} \cup \Sigma!_{c2} \\
\Sigma?_{c2} &= \{\text{Unlock}, \text{Vote Bob}\} \\
\Sigma!_{c2} &= \{\text{Bob:}i, \text{Vote Cast For: ``Alice''}, \\
&\qquad \text{Vote Cast For: ``Bob''}, \text{Vote Cast For: ``Charlie''}\} \\
s_{0\,c2} &= c2_a \\
\rightarrow_{c2} &= \{(c2_a, \text{Unlock}, c2_a), (c2_a, \text{Bob:}i, c2_a), \\
&\qquad (c2_a, \text{Vote Cast For: ``Alice''}, c2_a), \\
&\qquad (c2_a, \text{Vote Cast For: ``Bob''}, c2_a), \\
&\qquad (c2_a, \text{Vote Cast For: ``Charlie''}, c2_a), \\
&\qquad (c2_a, \text{Vote Bob}, c2_a)\}
\end{aligned}
$$

C2 and UI1 are composed to C1 (see Figure 6.15):

$$
\begin{aligned}
L_{c1} &= (S_{c1}, \Sigma_{c1}, s_{0\,c1}, \rightarrow_{c1}) \text{ with} \\
S_{c1} &= \{ca1_a\} \\
\Sigma_{c1} &= \Sigma?_{c1} \cup \Sigma!_{c1} \\
\Sigma?_{c1} &= \{\text{Unlock}, \text{Vote Bob}\} \\
\Sigma!_{c1} &= \{\text{Ready}, \text{Vote Cast For: ``Alice''}, \text{Vote Cast For: ``Bob''}, \\
&\qquad \text{Vote Cast For: ``Charlie''}\} \\
s_{0\,c1} &= ca1_a \\
\rightarrow_{c1} &= \{(c1_a, \text{Unlock}, c1_a), (c1_a, \text{Ready}, c1_a), \\
&\qquad (c1_a, \text{Vote Cast For: ``Alice''}, c1_a), \\
&\qquad (c1_a, \text{Vote Cast For: ``Bob''}, c1_a), \\
&\qquad (c1_a, \text{Vote Cast For: ``Charlie''}, c1_a), (c1_a, \text{Vote Bob}, c1_a)\}
\end{aligned}
$$

UB0 and C1 are composed to CU0 (see Figure 6.16):

$$
\begin{aligned}
L_{c1} &= (S_{c1}, \Sigma_{c1}, s_{0\,c1}, \rightarrow_{c1}) \text{ with} \\
S_{c1} &= \{c1_a\} \\
\Sigma_{c1} &= \Sigma?_{c1} \cup \Sigma!_{c1} \\
\Sigma?_{c1} &= \{\text{Unlock}\} \\
\Sigma!_{c1} &= \{\text{Vote Cast For: ``Alice''}, \text{Vote Cast For: ``Bob''}, \\
&\qquad \text{Vote Cast For: ``Charlie''}\} \\
s_{0\,c1} &= c1_a \\
\rightarrow_{c1} &= \{(c1_a, \text{Unlock}, c1_a), \\
&\qquad (c1_a, \text{Vote Cast For: ``Alice''}, c1_a), \\
&\qquad (c1_a, \text{Vote Cast For: ``Bob''}, c1_a), \\
&\qquad (c1_a, \text{Vote Cast For: ``Charlie''}, c1_a)\}
\end{aligned}
$$

In the last step, CU0 and AB0 are composed. Since the output alphabet of CU0 is restricted to { Vote Cast For: “Alice”, Vote Cast For: “Bob”, Vote Cast For: “Charlie” }, only $ab0_c$ with $c =$ “Alice”, $c =$ “Bob”, or $c =$ “Charlie” are reachable. Therefore, we can reduce AB0 to

$$
\begin{aligned}
L_{ab0} &= (S_{ab0}, \Sigma_{ab0}, s_{0\,ab0}, \rightarrow_{ab0}) \text{ with} \\
S_{ab0} &= \{ab0_a, ab0_b, ab0_c, ab0'_c, ab0''_c\} \\
\Sigma_{ab0} &= \Sigma?_{ab0} \cup \Sigma!_{ab0} \\
\Sigma?_{ab0} &= \{\text{Vote Cast For: “Alice”}, \text{Vote Cast For: “Bob”}, \\
&\qquad \text{Vote Cast For: “Charlie”}\} \\
\Sigma!_{ab0} &= \{\text{Unlock}\} \\
s_{0\,ab0} &= ab0_a \\
\rightarrow_{ab0} &= \{(ab0_a, \text{Unlock}, ab0_b), \\
&\qquad (ab0_b, \text{Vote Cast For: “Alice”}, ab0_{c'}), \\
&\qquad (ab0_b, \text{Vote Cast For: “Bob”}, ab0_c), \\
&\qquad (ab0_b, \text{Vote Cast For: “Charlie”}, ab0_{c''})\}
\end{aligned}
$$

Finally, C as the composition of CU0 and AB0 is computed as

$$
\begin{aligned}
L_c &= (S_c, \Sigma_c, s_{0\,c}, \rightarrow_c) \text{ with} \\
S_c &= \{c_a\} \\
\Sigma_c &= \Sigma?_c \cup \Sigma!_c \\
\Sigma?_c &= \{\} \\
\Sigma!_c &= \{\} \\
s_{0\,c} &= c_a \\
\rightarrow_c &= \{\}
\end{aligned}
$$

In difference to C from Section 6.2.1, this time the only resulting state $c_a$ is not only composed from the *success* final state $ab$“Bob” and a number of non-fatal states. The *fatal* final states $ab$“Alice” and $ab$“Charlie” are also part of the composed state $c_a$. Therefore, *fatal* final states are reachable, i.e. in the given model it is possible that the user votes for “Alice” or “Charlie” although he wanted to vote for “Bob.”

In this chapter, we have introduced a methodology to create hierarchical user and application models. This methodology allows to model HCI at any chosen level of granularity, down to the most basic operations. Creating local compositions and abstractions of components allows pervasive evaluation of models without suffering from state-space explosion.

# Chapter 7

# Integration with Hoare Logic

In previous chapters, we modeled the abstract behavior of applications using labeled transition systems, where nodes represent states of the application. For example, the basic version of the eVoting application model shown in Chapter 4 (Figure 4.6) contains the states "Unlocked", "Vote Cast 'Bob'", and "Vote Confirmed 'Bob'". Edges represent transitions from one state to the next. These states are abstractions of the actual system configurations, which are much richer in detail. (IO)LTS and temporal logics are suitable methods to specify the abstract behavior of concurrent components of a system. Model checking can be used to verify that an abstract model has the desired properties. For a pervasive specification of an application, it is also necessary to prove that a concrete program is a refinement of the abstract model. That requires a specification of a concrete program's procedures using pre- and post-conditions. Methods like Hoare logic (Hoare, 1969) and Dynamic Logic (Harel, 1984) can then be used verify the concrete implementation.

The pre-/post-condition-based specification and the state-based IOLTS/CTL methods have to be integrated in order to provide a unified methodology for the pervasive specification and verification of secure interactive systems. A relationship between the nodes, edges, and labels of the IOLTS on the one hand, and the pre-conditions, post-conditions, and procedure implementations on the other hand has to be established. The intuitive relationship between a IOLTS and program functionality is that nodes represent abstractions of program states, while pre- and post-conditions refer to properties of concrete states (and each condition defines a state set, namely the set of all states where the condition is true). Labeled edges between nodes represent program function calls and their input and output.

In the following definitions, we assume that a set $H$ of formulas is given that can be used as pre-/post-conditions and invariants in Hoare-style specifications and proofs, as well as a set $P$ of procedures used in concrete implementations and a set $M$ of messages sent and received by these procedures. For our purposes it

is not necessary to fix a particular logic and a programming language. In Part III, we show how our methodology has been used in the Verisoft project (Paul, 2005). In the Verisoft project, we used Isabelle/HOL (Schirmer, 2005) for $H$ and C0 (a subset of the C programming language (Leinenbach et al., 2005)) for procedures in $P$.

**Definition 7.1** (Pervasive model)**.** *Let $H$ be the set of all formulas that can be used as pre-/post-conditions and invariants. Let $P$ be the set of all procedures that take exactly one input and one output parameter. And let $M$ be the set of input/output potentially sent or received by procedures in $P$.*

*Then, a* pervasive model

$$PM = (L, inv, proc, msg)$$

*consists of*

- *an IOLTS*

  $$L = (S, \Sigma, s_0, \rightarrow_L) \ ,$$

- *a mapping*

  $$inv : S \rightarrow H$$

  *assigning to each state in $S$ a state invariant from $H$,*

- *a mapping*

  $$msg : \Sigma \rightarrow 2^M$$

  *associating labels of the IOLTS with actual procedure input/output,*

- *a mapping*

  $$proc : (S \times \Sigma?) \rightarrow P$$

  *from state/input pairs to procedures.*

Intuitively, the concrete implementation of an application is correct w.r.t. a pervasive model *PM* if

1. it is a refinement of the IOLTS part of *PM*, i.e., its control flow corresponds to the IOLTS,
2. its procedures are correct w.r.t. the state invariants of *PM*.

**Algorithm 3** Preliminary main event loop

```
1: repeat
2:    cmd := getKeystroke()
3:    updateScreen(cmdResult)
4:    cmdResult := execute(state,cmd)
5:    state := nextState(state,cmd,cmdResult)
6: until cmd = QUIT
```

The Hoare logic specification depends on a correct "execution" of the IOLTS representing the application. In the following, we take a first step towards bridging the gap between the abstract state-based model of an application and the specification of the concrete program's with pre-/post-conditions. We do this by providing a template for a main execution loop that "executes" the high level model by calling the procedures `updateScreen`, `getKeystroke`, `execute`, and `nextState`. A generic template for a main execution loop that executes an IOLTS is given in Algorithm 3. Please note that this algorithm is preliminary. It will be improved in Section 14.3.2. In this template, the procedure `nextState` implements the state transitions of the IOLTS. The actual state is stored in program variable `state`. Procedure `execute` implements execution of the procedures associated with labeled transitions.

Procedure `getKeystroke` gets the next keystroke, and procedure `updateScreen` shows the result of command execution on the screen. The actual implementation of these procedures depends on the concrete problem. We will give examples for an email client in Part III of this thesis.

The procedure `nextState` must guarantee that the application "executes" the IOLTS. It gets the old states as an input parameter and returns the new state as its result. The transition from the old to the new state must represent a valid state transition, i.e. there must exist a transition $\texttt{oldState} \xrightarrow{(\sigma?/\sigma!)} \texttt{result}$ in the IOLTS. The input command must be in the message set represented by $\sigma?$, and the result of the command execution must be in the message set represented by $\sigma!$. The transition relation may be non-deterministic.

**Definition 7.2** (Specification of procedure `nextState`)**.**

| | |
|---|---|
| ***context*** | $nextState(oldState, cmd, cmdResult)$ |
| ***pre*** | ***True*** |
| ***post*** | $\exists \sigma?, \sigma! : oldState \xrightarrow{(\sigma?/\sigma!)} result$ |
| | $\wedge\ cmd \in msg(\sigma?)$ |
| | $\wedge\ cmdResult \in msg(\sigma!)$ |

The procedure `execute` is called by the main execution loop to invoke the concrete procedures associated with edges of the application logic IOLTS. The command given by the user related to an IOLTS input symbol $\sigma?$ by function *msg*, and the procedure to be called is identified by the function *proc*. The output of the procedure call is related to an IOLTS output symbol $\sigma!$. The state invariant of the pre-state holds before `execute` is called, and the state invariant of the successor state must hold after `execute` returns.

**Definition 7.3** (Specification of procedure `execute`)**.**

| | |
|---|---|
| ***context*** | $execute(state,\ command)$ |
| ***pre*** | $inv(state)$ |
| ***post*** | $\exists\, \sigma?, \sigma!, s : state@pre \xrightarrow{(\sigma?/\sigma!)} s \wedge inv(s) \wedge$ $cmd \in msg(\sigma?) \wedge result \in msg(\sigma!)$ |

The specification of `execute` does not refer explicitly to the procedures associated with edges. It only guarantees that the invariants of states hold in its postcondition.

Next, we define the relationship between the function *proc* and `execute`. We start by defining *edge procedure correctness* for *proc*. Mapping *proc* associates edges with procedures. The result of function *proc* (i.e. the procedure to be called) depends on the current state of the application, and the input $\sigma?$ for which the input command $\texttt{cmd} \in \sigma?$. In the definition of a pervasive model (Definition 7.1), invariants are associated with states. Whenever the system is in a given state, the invariant of the state must be satisfied. Therefore, the precondition for a procedure call must be implied by the invariant of the state in which the procedure is called, and the postcondition of the procedure must imply the invariant of the succeeding state:

**Definition 7.4** (Edge procedure correctness)**.** *Let*

$$PM = (L, inv, proc, msg)$$

*be a pervasive model.* Edge procedure correctness *is guaranteed if for all values of* $\sigma?$*,* $s$ *and* $cmd$*, there exists* $\sigma!$ *and* $s'$ *such that*

$$\{inv(s) \wedge cmd \in msg(\sigma?)\}$$
$$proc(s, \sigma?)(cmd)$$
$$\{result \in msg(\sigma!) \wedge s \xrightarrow{(\sigma?/\sigma!)} s' \wedge inv(s')\}$$

From these definitions follows that `execute` is correct if `execute(state, cmd)` calls $proc(\texttt{state}, \sigma?)$ for all $\texttt{state} \in S$ and all commands `cmd` related to some $\sigma?$.

In Hoare-style specifications, only sequential aspects of a program are specified. We assume that changes in the configuration do only result from procedure calls specified in Hoare logic.

To conclude, the implementation of an application is correct w.r.t. a pervasive model if all of the following holds:

1. The main event loop follows Algorithm 4,
2. `nextState` satisfies the specification from Def. 7.2,
3. `execute` satisfies the specification from Def. 7.3,
4. the invariant $inv(s_0)$ of the initial state holds before any message is sent or received (i.e., immediately after initialization).

Following our approach, the correctness argument for an application is split into three parts:

- The high level IOLTS specification guarantees the desired properties (i.e., it has to satisfy the requirements).
- The main execution loop follows the template given in Algorithm 4, `nextState` implements the particular IOLTS, and `execute` call s the procedures associated with labeled edges in the IOLTS.
- The pervasive model satisfies Edge procedure correctness (Definition 7.4).

**Differences to Software Model Checking.** Our approach to use model checking for actual C code differs from software model checking approaches like the ones used in the SLAM project (Ball and Rajamani, 2001) or CEGAR, which is implemented in the MAGIC tool (Chaki et al., 2004). Software model checking reduces the state space of an actual implementation in order to apply model checking techniques to guarantee program properties. These techniques allow to find certain classes of errors in existing programs. In difference to this, our approach provides a methodology for the specification of applications which takes both sequential aspects of the program and parallel aspects into account. In our approach, the state space of a program is not automatically reduced in order to make it suitable for model checking. We expect the system designer to provide an explicit model both of the high-level system design (used with model checking), and the low-level design (used with Hoare logic). By defining constraints

on these design steps, we are able to provide a *pervasive* method for the formal specification and verification of both high-level temporal properties of a system, and low-level sequential properties.

# Chapter 8

# Summary

We laid the foundations of a methodology for formalizing, analyzing, and verifying user interfaces and human-computer interaction under computer security aspects. The main contributions of this part are a formal semantics for an extended version of GOMS, a generic user and application model suitable for the pervasive specification of human-computer interaction, and the integration of temporal specifications based on IOLTS/CTL with Hoare-style procedure specifications.

- We have introduced a formal semantics for GOMS models describing user behavior, which is based on input/output labeled transition systems (IOLTS).

- We showed how the component-based formalization of GOMS can be augmented with components modeling the user's assumptions about the application. This allows to model HCI both in absence and in presence of human errors.

- The method used to formalize GOMS models and the user's assumption can be applied to model the application as well. Combining all three components leads to a complete model of human-computer interaction suited for automated reasoning.

- We have introduced a methodology to formally describe hierarchical user interfaces. This makes the pervasive modeling of all aspects of user interface security possible.

- Component specifications based on IOLTS abstract from the actual program code. We developed a methodology integrating the specification and verification of high-level application behavior using IOLTS and temporal logic with specification and verification of low-level application behavior using

Hoare-style pre-/postconditions. This makes pervasive verification of applications possible.

We have developed a generic formal model of human-computer interaction with security critical applications. This formal model is the base for a systematic formalization of user interface security requirements. Our methodology can be used both for system design and for the analysis of properties of existing systems. It is applicable both to restricted specialized system as well as to generic, off-the-shelf systems. In the Verisoft project (`http://www.verisoft.de`), this approach has been used to prove human-computer interaction security of an email client application in the context of a pervasively verified computer system.

# Part II

# Formalization of HCI Security

# Chapter 9

# System Model

## 9.1 Messages

The methodology introduced so far allows to describe the internal behavior of components. For the formal specification of human-computer interaction, it is also necessary to explicitly describe properties of the messages exchanged between components, i.e. properties of the communication protocol. A wide range of formal methods are used in protocol analysis. Meadows (2003) gives an overview of formal methods used for cryptographic protocol analysis. For cryptographic protocols, the Dolev-Yao model (Dolev and Yao, 1981) and its various variants are widely used. Burrows et al. (1989) developed BAN logic, a logic for the description of the belief's of message agents about their communication partners and about the messages exchanged between them. BAN logic is decidable, and automated reasoning tools for BAN logic are available (Brackin, 1998). Other approaches use the standardized formal description techniques (FDTs) Estelle, SDL and LOTOS (Turner, 1993). Model checkers like FDR and theorem provers like Isabelle have been used for cryptographic protocol analysis (Lowe, 1996; Paulson, 1998).

Temporal logic based methods are rarely used for protocol specification, because temporal logic has no means to identify unique messages in a stream, and components are not composable, i.e. in order to guarantee the correctness of a specification, all component specifications must be available. It is not possible to evaluate the correctness of components independent of each other. Jmaiel (1994) introduces a method to overcome both weaknesses: By "coloring messages" unique messages can be identified. By introducing communication channels and the semantics of operators on channels, composability of components is achieved. In this work, we have adapted this idea to our approach for modeling HCI. Our approach differs from Jmaiel's in two ways. While Jmaiel uses LTL, we

use CTL, because CTL is more suitable for the formal description of HCI security requirements. Jmaiel defines the semantics of transition systems via traces of messages. Our definition of transition system semantics is based on the states of the transition system. We associate transition system states with the messages that lead into the states. This allows us to use the same definition of transition system semantics both for the internal working of the components (white box view), and the communication behavior of the components (black box view). This way, it becomes possible to apply our approach to individual components, and to define properties of communication protocols in our approach.

In Section 4.1, linear and parallel composition (Definitions 4.3 and 4.4) have been used in order to deduce properties of the system. An example for this approach has been given in Section 6.2.1. In order to deduce properties of the system, a complete specification of all components was required. We would, however, like to describe components separate from each other by describing their input/output behavior by logical formulae. In order to describe the messages sent and received by a component, we have to define logical propositions that hold whenever a message is send or received. The IOLTS semantics defined in Section 4.1 are based on the state of the IOLTS. Valuation function $\lambda$ takes the current state of the IOLTS as its argument. In order to describe components by the messages input and output of the component, we give an alternative definition of IOLTS semantics based on traces instead of paths (see Definition 4.6 in Chapter 4).

**Definition 9.1** (Trace Semantics)**.** *Given an IOLTS $L = (S, \Sigma, s_0, \rightarrow)$, a domain $D$, and a set of interpretations $I$ a* trace valuation $\lambda$ *is a mapping from $\Sigma$ to $I$. $L, \lambda, \sigma_0 \models \phi$ denotes that $\phi$ holds in state $\sigma_0$ with valuation function $\lambda$. $L, \lambda, x \models \phi$ denotes that $\phi$ holds for all paths $x = \langle \sigma_0, \sigma_1, \ldots \rangle$ with valuation function $\lambda$. $\lambda$ is defined inductively as as follows:*

$L, \lambda, \sigma_0 \models p(t_1, \ldots, t_n)$ *if* $(I(t_1), \ldots, I(t_n)) \in I(p)$ *with* $I = \lambda(\sigma_0)$

$L, \lambda, \sigma_0 \models \neg\, \phi$ *if not* $L, \lambda, \sigma_0 \models \phi$

$L, \lambda, \sigma_0 \models \phi \wedge \psi$ *if* $L, \lambda, c \models \phi$ *and* $L, \lambda, \sigma_0 \models \psi$

$L, \lambda, \sigma_0 \models \phi \vee \psi$ *if* $L, \lambda, \sigma_0 \models \phi$ *or* $L, \lambda, \sigma_0 \models \psi$

$L, \lambda, \sigma_0 \models \forall x.\phi$ *if* $L, \lambda, \sigma_0 \models \phi_{[x/y]}$ *for all* $y \in D$

$L, \lambda, \sigma_0 \models \exists x.\phi$ *if* $L, \lambda, \sigma_0 \models \phi_{[x/y]}$ *for at least one* $y \in D$

$L, \lambda, x \models \phi$ *if* $L, \lambda, \sigma_0 \models \phi$

$L, \lambda, x \models \boldsymbol{A}\phi$ *if* $L, \lambda, x \models \phi$ *for all paths $x$ in $L$ starting with $\sigma_0$*

$L, \lambda, x \models \boldsymbol{E}\phi$ *if* $L, \lambda, x \models \phi$ *for at least one path $x$ in $L$ starting with $\sigma_0$*

$L, \lambda, x \models \boldsymbol{X}\phi$ *if* $L, \lambda, x^1 \models \phi$

$$L, \lambda, x \models \phi \mathbf{U} \psi \text{ if (a) } L, \lambda, \sigma_0 \models \psi$$
$$\text{or (b) there is some } i \geq 1 \text{ s.t. } L, \lambda, x^i \models \psi$$
$$\text{and } L, \lambda, x^k \models \phi \text{ for all } 0 \leq k < i$$

$$L, \lambda, x \models \mathbf{G}\phi \text{ if } L, x^i \models \phi \text{ for all } i \geq 0$$

$$L, \lambda, x \models \mathbf{F}\phi \text{ if } L, x^i \models \phi \text{ for some } i \geq 0$$

Temporal logic statements about messages become possible if valuation function $\lambda$ provides *explicit message passing*. We define *message predicates* in the same way we defined state predicates (see Definition 4.9 in Section 4.2):

**Definition 9.2** (Message Predicate)**.** *Let $L = (S, \Sigma, s_0, \rightarrow)$ be an IOLTS. Let $\lambda$ be a trace valuation function. The model contains* message predicates *if the trace valuation function $\lambda$ has the following properties:*

$$\textit{messagePreds}(L, \lambda) \equiv L, \lambda, \sigma \models \sigma$$
$$L, \lambda, \sigma' \not\models \sigma \quad \text{if } \sigma \neq \sigma'$$

With these definitions, it becomes possible to prove temporal logic statements about message input and output of the application. For example, we can show that it is possible that the component from Figure 4.6 (Chapter 4) never receives "CancelVote" immediately followed by "ConfirmVote". Appendix A.3 shows the changes to the file given in Appendix A.1 in order to check this property with NuSMV.

Now, we introduce the approach of Jmaiel (1994) for composable component definitions. With this approach, it becomes possible to prove properties of components independently of other components. When composing components, the output symbols of one component are identical to the input symbols of the other component and both systems are run synchronously. Jmaiel (1994) calls the (unidirectional) connection between systems *channels*. In Jmaiel's approach, channels are named. For two components $A$ and $B$, connected by a communication channel $X$, transfer of a message $m$ is represented by predicates $[A \textbf{ snd } m \textbf{ on } X]$, $[X \textbf{ xmt } m]$, and $[B \textbf{ rcv } m \textbf{ on } X]$. We assume that communication channels are fixed and messages are not lost on transport[1], therefore

$$[A \textbf{ snd } m \textbf{ on } X] \equiv [X \textbf{ xmt } m] \equiv [B \textbf{ rcv } m \textbf{ on } X]$$

if $A$ and $B$ are connected by $X$. In Jmaiel's approach, $[S \textbf{ xmt } m]$ are atomic propositions for all components $S$ and for all messages $m$. We link this approach to our modeling method by defining a relationship between input/output symbols and message transfers:

[1] Lossy channels can by modeled as lossy components in between two communication channels.

**Definition 9.3 ( xmt -syntax).** *Let $L = (S, \Sigma, s_0, \rightarrow)$ be an IOLTS. $L$ is in* **xmt** -syntax *if*

- *the set of input and output symbols is split into $n$ sets $\Sigma_0, \ldots, \Sigma_n$ with $\Sigma = \bigcup \Sigma_i$ and $\bigcap \Sigma_i = \varnothing$.*
- *there exist channel names $C_0, \ldots, C_n$ and a set of messages $M$ such that $[C_i$ **xmt** $m] \in \Sigma_i$ for all $m \in M$ and $C_i$.*

We define an equivalence relationship on the set of messages. Relationship "=" indicates that two messages have the same contents:

**Definition 9.4.** *For all message $m$ and $m'$, $m = m'$ iff $m$ and $m'$ have identical contents.*

The definition of "=" is not sufficient for the definition of communication protocols, because it does not allow to discriminate between message with the same content, i.e. it does not allow to uniquely identify messages on a stream, as shown by Koymans (1992). Jmaiel uses "colors" to overcome this problem. Each message has a distinct color. He uses a hierarchy of indexed congruence relationships on the set of colors $\sim_0, \sim_1, \ldots, \sim_n$ forming an inclusion chain $\sim_n \subset \sim_{n-1} \subset \ldots \subset \sim_0$ for this, i.e. the following axioms hold:

**Definition 9.5** (Congruence Axioms)**.**

$$
\begin{array}{ll}
m \sim m & \textit{for all} \sim \in \{\sim_0, \ldots, \sim_n\} \\
m_1 \sim m_2 \wedge m_2 \sim m_3 \rightarrow m_1 \sim m_3 & \textit{for all} \sim \in \{\sim_0, \ldots, \sim_n\} \\
m_1 \sim m_2 \rightarrow m_2 \sim m_1 & \textit{for all} \sim \in \{\sim_0, \ldots, \sim_n\} \\
m \sim_{i+1} m' \rightarrow m \sim_i m' &
\end{array}
$$

We define that all incoming messages are distinct in respect to relationship $\sim$:

**Definition 9.6** (Distinct Coloration)**.**

$$
\begin{array}{l}
\textit{distCol}(s, \sim) \equiv \\
\forall\, m, m'.[s \textbf{ xmt } m] \wedge \textbf{EXEF}[s \textbf{ xmt } m'] \rightarrow m \not\sim m'
\end{array}
$$

As an example for this methodology, we give the definition of a screen component. The component introduced here is part of the generic system model described in Section 9.2. The screen component takes input from an application and presents the data to the user. A screen is an "asynchronous" component. Once it received some input, it will continuously output it until different input is provided.

Let $d$ be an output device with input channel $s$ and output channel $r$. The property of distinctively colored messages makes it possible to uniquely identify all message ever sent on a path:

$$\mathit{distCol}(s, \sim_0)$$

Incoming messages are output by the component. For every incoming message, there is an outgoing message from the same congruence class in respect to $\sim_0$:

$$\forall\, m, m'.[s\ \mathbf{xmt}\ m] \rightarrow \mathbf{AX}[r\ \mathbf{xmt}\ m'] \wedge m \sim_0 m'$$

Finally, we define that the same message is continuously output until the screen receives new input:

$$\forall\, m', m''.(\forall\, m.\neg[s\ \mathbf{xmt}\ m]) \wedge [r\ \mathbf{xmt}\ m'] \rightarrow \mathbf{AX}[r\ \mathbf{xmt}\ m''] \wedge m' \sim_0 m''$$

This leads to the following definition in one formula:

**Definition 9.7** (Screen)**.** *Let $d$ be an output device with input channel $s$ and output channel $r$. $d$ is a* Screen *if*

$$\begin{aligned}&\mathit{distCol}(s, \sim_0) \wedge \forall\, m, m'.[s\ \boldsymbol{xmt}\ m] \rightarrow \boldsymbol{AX}[r\ \boldsymbol{xmt}\ m'] \wedge m \sim_0 m' \\ &\wedge \forall\, m', m''.(\forall\, m.\neg[s\ \boldsymbol{xmt}\ m]) \wedge [r\ \boldsymbol{xmt}\ m'] \rightarrow \\ &\qquad \boldsymbol{AX}[r\ \boldsymbol{xmt}\ m''] \wedge m' \sim_0 m''\end{aligned}$$

We assume that all incoming messages of the screen component are distinct in respect to relationship $\sim_0$. Outgoing messages are not distinct in this respect. If one wants to compose screen with a subsequent component, it is desirable to be able to discriminate between outgoing messages as well. For this, a new congruence relationship $\sim_1$ can be defined as $\mathit{distCol}(d, \sim_1)$. With these definitions, we can uniquely identify incoming messages of the first component via relation $sim_0$, and uniquely identify incoming message of the second component via relation $\sim_0$.

## 9.2 Environment

The generic system model described in this chapter serves as a blueprint for the definition of application models, user models, and security requirements. The requirements for the application and the user model are as follows:

1. Be as generic as possible.

   We are not interested in special purpose applications or in users with a specific goal in mind. The goal is to find models fitting large classes of applications and users.

2. Be as simple as possible.

   The methodology for guaranteeing secure HCI should be suitable for automated reasoning techniques. Therefore, there should be as few components with as few states as possible in the model.

3. Abstract from everything not relevant for security

   Formal methods in HCI are usually used for usability studies. In these studies, a sophisticated an highly specialized user model is required. We use formal methods to model HCI security. This allows to reduce the complexity of the application and the user's mental model to those aspects which are relevant from a security point of view. It becomes possible to use simple and generic user and application models and we can avoid the complexity of models focusing on functionality and usability.

4. Build upon established methods

   The system model should benefit from previous work on formal models for human-computer interaction.

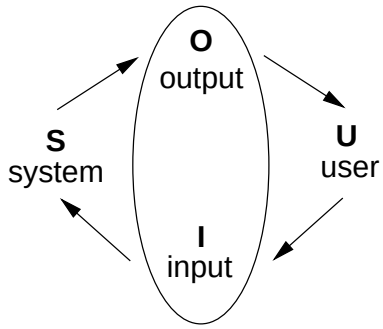


Figure 9.1: Abowd and Beale's Interaction Framework (adapted from (Dix et al., 1998))

In general, human-computer interaction can be described as a dialog between two parties: A user and an application. Interactions can be described as traces of message passing between user and application. Our template for a generic system model follows Abowd and Beale's *Interaction Framework*. Abowd and Beale (1991) describe human computer interaction as a communication process between four parties: The system ($S$), a user ($U$), an input interface ($I$), and an output interface ($O$), as shown in Figure 9.1. The interaction framework model of HCI describes all relevant parts of human-computer interaction while providing an abstraction that is both suitable to encompass large classes of applications and to describe security-relevant properties of human-computer interaction.

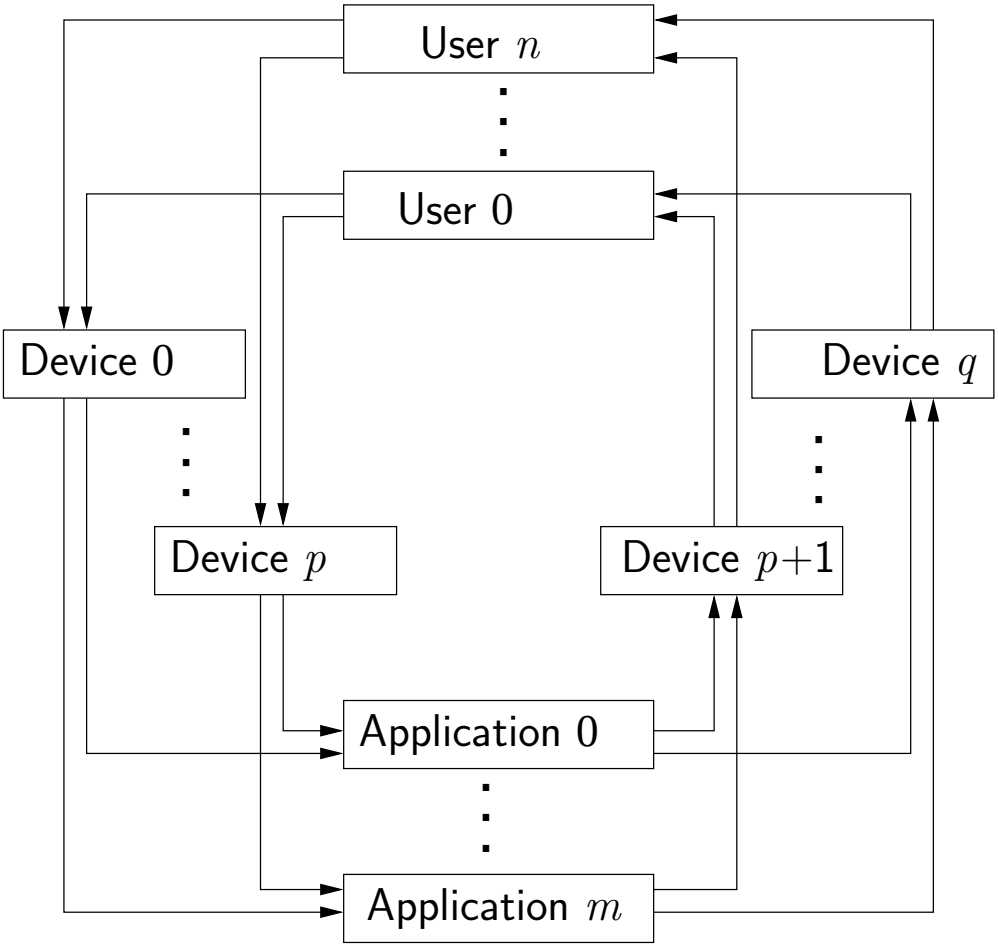


Figure 9.2: Generic System Model

Abowd and Beale's interaction framework models a system, a user, and the interfaces between them. In modern HCI environments, this model is only a special case of a more general model of human-computer interaction. Our generic model models a generic scenario, where a user may interact with a number of applications (the system is multi-tasking), and a number of users may have access to the system (the system is multi-user.) This leads to the interaction model depicted in Figure 9.2.

There is a number of users, a number of applications, a number of devices for user input (e.g. a keyboard, a mouse, a dedicated keypad for PIN entry), and a number of devices for application output (e.g. screen, loudspeaker, dedicated screen for PIN entry).

In our environment model, all users can send messages to all input devices, all applications can receive messages from all input devices, all applications can send messages to all output devices, and all users can receive messages from all output devices. A communication path consists of a sender, a receiver, a device, and two communication channels; one communication channel from the sender to the device, and one communication channel from the device to the receiver. We refer to these channels as $c_{(s,d)}$ and $c_{(d,r)}$, where $c$ refers to the channel, $s$ refers to the sender, and $r$ refers to the receiver.

**Definition 9.8** (Environment)**.** *Let $U$ be the set of users with $u = (s_u, \Sigma_u, s_{0u}, \rightarrow_u)$ and $\Sigma_u = \Sigma?_u \cup \Sigma!_u$ for all $u \in U$. Let $A$ be the set of applications with $a = (s_a, \Sigma_a, s_{0a}, \rightarrow_a)$ and $\Sigma_a = \Sigma?_a \cup \Sigma!_a$ for all $a \in A$. Let $D$ be the set of devices with $d = (s_d, \Sigma_d, s_{0d}, \rightarrow_d)$ and $\Sigma_d = \Sigma?_d \cup \Sigma!_d$ for all $d \in D$. Let the set of devices be divided into a set of input devices $I$ and output devices $O$: $D = I \cup O$ and $I \cap O = \varnothing$. Let $M$ be a set of messages. The* environment *env$(U, A, I, O, M)$ is defined a*

- *All elements of U,A,I, and O are in **xmt** -syntax with*
  - $\Sigma?_{u'} = \{[c_{(o,u')}$ ***xmt*** $m] \mid o \in O, m \in M\}$ *for all* $u \in U$.
  - $\Sigma!_{u'} = \{[c_{(u',i)}$ ***xmt*** $m] \mid i \in I, m \in M\}$ *for all* $u \in U$.
  - $\Sigma?_{a'} = \{[c_{(i,a')}$ ***xmt*** $m] \mid i \in I, m \in M\}$ *for all* $a \in A$.
  - $\Sigma!_{a'} = \{[c_{(a',o)}$ ***xmt*** $m] \mid o \in O, m \in M\}$ *for all* $a \in A$.
  - $\Sigma?_{i'} = \{[c_{(u,i')}$ ***xmt*** $m] \mid u \in U, m \in M\}$ *for all* $i \in I$.
  - $\Sigma!_{i'} = \{[c_{(i',a)}$ ***xmt*** $m] \mid a \in A, m \in M\}$ *for all* $i \in I$.
  - $\Sigma?_{o'} = \{[c_{(a,o')}$ ***xmt*** $m] \mid a \in A, m \in M\}$ *for all* $o \in O$.
  - $\Sigma?_o = \{[c_{(o',u)}$ ***xmt*** $m] \mid u \in U, m \in M\}$ *for all* $o \in O$

We refer to communication paths by triples $(s, d, r)$. In our model, communication paths exists from the user via an input device to the application, and from the application via an output device to the user

**Definition 9.9** (Path)**.** *Let env*$(U, A, I, O, M)$ *be an environment. The set of paths* $t($*env*$(U, A, I, O, M))$ *in the environment is defined as*

$$(\textit{paths}(U, A, I, O, M)) \equiv \begin{array}{l} \{(u, i, a) \mid u \in U, i \in I, a \in A\} \\ \cup\{(a, o, u) \mid u \in U, o \in O, a \in A\} \end{array}$$

With the environment model defined in Definition 9.8, we can describe the following typical scenarios in a multi-user, multi-tasking environment:

**Single user, single application** The traditional scenario of the interaction of one user with one application is modeled by restricting the number of users and applications to 1. This scenario is suitable to model typical dedicated devices for secure human-computer interaction, like voting computers, where authorized access to the machine is secured by external means:

> *singleUserSingleAppEnv* $\equiv$
> *env*$(U, A, I, O, M)$ and $|U| = 1$ and $|A| = 1$

**Single users, multiple applications** In an environment where each workplace is accessible to one user only (e.g. an office environment where physical locks prevent employees to access the workspaces of other employees), but the user may interact with a number of applications, the number of users ($n$) is limited to 1, but multiple applications ($m$) may access the keyboard and the screen.

> *singleUserSingleAppEnv* $\equiv$
> *env*$(U, A, I, O, M)$ and $|U| = 1$

**Multiple users, single or multiple applications** In the most open scenario, multiple users have access to the computer system, which may run a single or multiple applications. One can further distinguish between scenarios where one user has access to the keyboard, but multiple users have access to the screen (one user is using the computer, but others may look over his shoulders), and full multi-user access to the system.

Next, we can define the system model used to describe HCI security by combining the environment model with the user and application model. A system model is an environment model with a distinguished user ($u$) and a distinguished

application ($a$). These are the user and application that we are modeling under security aspects, while all other users and applications are third parties.

In Section 4.2, we defined core application predicates (Definition 4.10) and core user predicates (Definition 4.11). The purpose of these definitions was to define a fixed set of predicates that are used in all models of users and applications. These generic predicates provide a generic framework for the formal statements of application and user behavior. In the same way, we define core message predicates. The core message predicates ensure that all incoming messages of application input and output devices are distinctively colored. Distinct coloration is required for the discrimination between messages passed by a component. Secondly, the core message predicates define predicates *secret* and *legitimate*. These predicates respectively indicate if a message is secret, i.e. if it may be received by authorized parties only, and who is legitimated to send and receive a message.

**Definition 9.10** (Secret and Legitimate Predicates)**.** *Let env*$(U, A, I, O, M)$ *by an environment. Let* $L = (S, \Sigma, s_0, \rightarrow)$ *be an IOLTS modeling an application. Let* $\lambda$ *be a valuation function. The model contains* secret and legitimate predicates *if secret and legitimate are in* $P$ *and the valuation function* $\lambda$ *has the following properties:*

*secretLegitimatePreds*$(L, \lambda) \equiv$

| | |
|---|---|
| $L, \lambda, s \models$ *secret*$(s, d, r, m)$ | *if* $s$ *needs legitimization when sending* $m$ *to* $r$ *via device* $d$ |
| $L, \lambda, s \models$ *legitimate*$(s, d, r, m)$ | *iff* $s$ *is legitimized to send message* $m$ *to* $r$ *via channel* $d$ |

**Definition 9.11** (System Model)**.** *Let env*$(U, A, I, O, M)$ *be an environment. system*$(U, A, I, O, M, u, a, \lambda)$ *is a system model iff*

- $u \in U$
- $a \in A$
- *coreAppPreds*$(a, \lambda)$
- *coreUserPreds*$(u, \lambda)$
- *secretLegitimatePreds*$(L, \lambda)$
- *For all users* $x$ *and input channels* $i$*: distCol*$(c_{(x,i)}, \sim_0)$
- *For all applications* $y$ *and input channels* $o$*: distCol*$(c_{(y,i)}, \sim_0)$

# Chapter 10

# The Common Criteria

In this chapter, we investigate existing security catalogs for their applicability to user interface security. Our goal is to adapt the security requirement definitions of the security catalogs to user interface security. By combining the formal concepts developed from the Common Criteria with the formalizations of the generic concepts of information security developed in Chapter 11, we create a comprehensive and systematic definition of human-computer interaction security.

## 10.1 Introduction to CC

Security catalogs are (more or less structured) collections of security requirements. They define properties of secure systems and provide a valuable resource for the evaluation of security threats. The best known security book is the "Department of Defense Trusted Computer System Evaluation Criteria" from 1985, also known as the "Orange Book" (DoD 5200.28-STD). Although seminal, the Orange Book is obsolete nowadays. Various national and international organizations adapted and enhanced computer security catalogs in the following decades, leading to the Common Criteria for Information Technology Security Evaluation (CC) (Common Criteria Evaluation Board (CCEB), 2006). The Common Criteria are the most important attempt to formalize information system security. The Common Criteria have been developed by the governmental computer security organizations of Canada, France, Germany, The Netherlands, The United Kingdom, and The United States. It has become the international standard ISO/IEC 15408:1999 and is one of the most comprehensive computer security catalogs. On the downside, the Common Criteria provide a large number of functions to pick from, without too much structure.

Security guidelines like the Common Criteria do not only define criteria for security functionality, but also criteria for the design and evaluation process of

| | |
|---|---|
| TOE | target of evaluation (TOE) — a set of software, firmware and/or hardware possibly accompanied by guidance. |
| TSF | TOE Security Functionality (TSF) — a set consisting of all hardware, software, and firmware of the TOE that must be relied upon for the correct enforcement of the SFRs. |
| SFR | security functional requirement |
| PP | Protection Profile (PP) — an implementation-independent statement of security needs for a TOE type. |
| SFP | security function policy (SFP) — a set of rules describing specific security behavior enforced by the TSF and expressible as a set of SFRs. |

Table 10.1: Glossary of Common Criteria terminology and abbreviations (Common Criteria Evaluation Board (CCEB), 2006, pages 16–18, 27)

applications. The highest evaluation level is EAL 7. EAL 7 requires a formal presentation of the functional specification and high-level design, and a formal and semi-formal demonstration of the correspondence between the high-level and the low-level design. The correspondence with the actual implementation is shown by extensive testing only. In this chapter, the parts of the Common Criteria relevant to human-computer interaction are identified and formalized in CTL.

The Common Criteria define the following classes:

- Class FAU: Security audit
- Class FCO: Communication
- Class FCS: Cryptographic support
- Class FDP: User data protection
- Class FIA: Identification and authentication
- Class FMT: Security management
- Class FPR: Privacy
- Class FPT: Protection of the TSF
- Class FRU: Resource utilisation
- Class FTA: TOE access
- Class FTP: Trusted path/channels

Table 10.1 contains a glossary of Common Criteria terminology and abbreviations used in this thesis. In this chapter, we identify those elements of the Common Criteria which are relevant to HCI security and formalize these. In order to link

the formal definitions of Common Criteria concepts developed in this chapter to the generic IT security requirements defined in the next chapter, we structure the classes of the Common Criteria along the lines of Paths and Identification (Section 10.2) and Privacy and Confidentiality (Section 10.3). Only classes relevant to HCI security are taken into consideration. We start with a set of core definitions. The core definitions are subsequently used to assign formal definitions to Common Criteria concepts.

From the classes provided by the Common Criteria, the three classes Security Audit (FAU), Cryptographic Support (FCS) and Security Management (FMT) are not covered in this thesis, because they are not directly related to user interface security. While Security Management (FMT), i.e. assignment and revocation of security attributes, is not subject of this work, it can be modeled by defining temporal properties on legitimate communication (see Section 10.3).

From the other classes, the most important one for HCI security is class FTP, because it defines security requirements for communication channels between parties. In Section 10.2, we define the core concepts for the formalization of FTP and the related classes FIA and FTA. We provide formal definition of the concept of a trusted path in Definition 10.1. FIA requirements are defined in respect to the identification of communicating parties in HCI and authentication procedures. Class FTA defines restrictions on the number of concurrent sessions per user and locking of devices is formalized.

While classes Resource Utilisation (FRU) and Communication (FCO) are also relevant for the definition of secure communication paths between the user and an application, they are not addressed in Section 10.2. Resource utilisation and resource allocation are already specified in the context of class FTA_MCS and FTA_SSL. Fault tolerance of I/O devices is discussed in the formalization of class FTA. For the access to I/O resources (FRU_PRS and FRU_RSA) sophisticated formal models of distribution and prioritizing shared resources are not required in the context of HCI security, because typical user I/O resources like keyboard, mouse, and screen can inherently be used by one user only at a time. Class Communication (FCO) deals with “assuring the identity of a party participating in a data exchange” (Common Criteria Evaluation Board (CCEB), 2006, part 2, page 43). We cover the identification of communicating parties in Section 10.2

Based on the core definitions of trusted paths, the relevant concepts of classes User data protection (FDP) and Privacy (FPR) are formalized in Section 10.3. In the formalization of class Protection of the TSF (FPT), the core concepts required for the definition of integrity and availability constraints are defined.

## 10.2 Paths and Identification (FTP, FIA, FTA, FRU, FCO)

The first group of classes addresses the definition of communication paths, the establishment, maintenance and closing of communication paths, and the identification of communicating parties. The core of these classes is class FTP, defining trusted paths and channels. Class Identification and Authentication (FIA) provides requirements for establishing authenticity of communication parties. Class TOE access (FTA) defines requirements for establishing and maintaining communication channels. Class Resource Utilisation (FRU) defines requirements for allocation of resources. In the context of HCI, we are interested in resources for the communication between the user and an application only. Class Communication (FCO) defines requirements for non-repudiability of origin and receipt of messages.

We start by developing a set of underlying core definitions in Section 10.2.1. These core definitions are used in Section 10.2.2 in the formal definition of Common Criteria concept definitions, and in Chapter 11 as a common basis for Common Criteria definitions and the generic information security concepts of Confidentiality, Integrity, and Availability.

Each of the definitions of the core concepts $C(\ldots)$ is accompanied by a definition of *asmC*$(x, \ldots)$, indicating if entity $x$ assumes that $C(\ldots)$ is true. These definitions become important when we model the beliefs of the user about the state of the system.

### 10.2.1 Core Definitions

In this chapter we develop formal definitions of communication paths. The core concept definitions build upon the generic system model from Chapter 9. In Common Criteria subclass FTP_TRP, a trusted path is defined as

> "FTP_TRP.1.1: The TSF shall provide a communication path between itself and [selection: remote, local] users that is logically distinct from other communication paths and provides assured identification of its end points and protection of the communicated data from modification or disclosure." (Common Criteria Evaluation Board (CCEB), 2006, part 2, page 171–172)

This is translated to the following formal definition:

**Definition 10.1** (Trusted Path)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path. The path is trusted, if the parties are authenti-*

*cated (each party knows the identity of the other party), and messages are neither modified nor leaked:*

$$\begin{aligned} \textit{trusted}((s,d,r)) \quad &\equiv \quad \textit{authenticated}((s,d,r)) \wedge \neg \textit{leaks}((s,d,r)) \\ &\quad\ \wedge \neg \textit{modifies}((s,d,r)) \end{aligned}$$

Predicate *asmTrusted* is introduced to formalize if a party assumes that a path is trusted:

**Definition 10.2** (Assumptions about Trusted Path)**.**
*Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path. Predicate asmTrusted*$(x, (s, d, r))$ *holds whenever* $x$ *assumes that property trusted (Definition 10.1) holds.*

The definition of *trusted* makes use of the predicate *authenticated*. A channel is authenticated if both the assumptions of the sender about the receiver, and the assumptions of the receiver about the sender are correct. We formalize these assumptions by predicate *asmIdentity*$(x, (s, d, r), s', d', r')$, indicating that user or application $x$, that the channel $(s, d, r)$ connects $s'$ via $d'$ to $r'$. The assumption of $x$ is correct if $s = s'$, $d = d'$, and $r = r'$.

**Definition 10.3** (Assumptions about Identities)**.**
*Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path. asmIdentity*$(x, (s, d, r), s', d', r')$ *holds if* $x$ *assumes that path* $(s, d, r)$ *connects* $s'$ *to* $r'$ *via device* $d'$*.*

A channel is successfully authenticated if both the assumptions of the sender about the receiver, and the assumptions of the receiver about the sender are correct. For a sending party $s$ and a receiving party $t$, this is the case when both *asmIdentity*$(s, (s, d, r), s, d, r)$ and
*asmIdentity*$(t, (s, d, r), s, d, r)$ hold:

**Definition 10.4** (Authentication)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path.*

$$\begin{aligned} \textit{authenticated}((s,d,r)) \quad &\equiv \quad \textit{asmIdentity}(s, (s,d,r), s, d, r) \\ &\quad\ \wedge \textit{asmIdentity}(t, (s,d,r), s, d, r) \end{aligned}$$

A channel leaks messages if messages may end up with a non-intended receiver. The definition of a system model (Definition 9.11), require distinct coloration (Definition 9.6) for all messages on input channels of the user and the application, i.e. for any two messages $m$ and $m'$ on any of the input channels,

$m \not\sim m'$. If a message $m'$ on an output channel has the same color as a message $m$ on an input channel ($m \sim m'$), it is the same messages. We use these properties in the definition of leaking. On a path $(s, d, r)$, all messages send on channel $c_{(s,d)}$ should by received on channel $c_{(d,r)}$. The path is leaking if a message send on $c_{(s,d)}$ may be received on a channel $c_{(d',r')}$ with $d \neq d'$ or $r \neq r'$:

**Definition 10.5** (Leaks)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path. The path* leaks *if*

$$\begin{aligned} \textit{leakes}((s,d,r)) \quad \equiv \quad & \exists\, m, m', d', r'. \\ & [c_{(s,d)}\ \textbf{\textit{xmt}}\ m] \wedge \textbf{\textit{EF}}[c_{(d',r')}\ \textbf{\textit{xmt}}\ m'] \\ & \wedge\, m \sim m' \wedge (d \neq d' \vee r \neq r') \end{aligned}$$

**Definition 10.6** (Assumptions about Leaks)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path. Predicate asmLeak*$(x, (s, d, r))$ *holds whenever* $x$ *assumes that property leaks (Definition 10.5) holds.*

Messages on a channel may be modified if the channel introduces messages (messages not sent by the sender are received by the receiver), if messages change during transport, if the order of messages may change, or if messages may get duplicated.

**Definition 10.7** (Modifies)**.**

$$\begin{aligned} \textit{modifies}((s,d,r)) \quad \equiv \quad & \textit{intro}((s,d,r)) \vee \textit{duplicates}((s,d,r)) \\ & \vee\, \textit{changes}((s,d,r)) \vee \textit{mixes}((s,d,r)) \end{aligned}$$

Concept *intro* describes that new messages may get introduced into a channel, either by a third party, or by spontaneous creation in the device. On a channel $(s, d, r)$, messages are introduced if there exists a message such that message $m$ is received, but no message $m'$ of the same color has been sent before:

**Definition 10.8** (Introduction of Messages)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path. New messages are created on path* $(s, d, r)$ *if*

$$\begin{aligned} \textit{intro}((s,d,r)) \quad \equiv \quad & \exists\, m. \forall\, m'. \\ & \textbf{\textit{A}}(\neg([c_{(s,d)}\ \textbf{\textit{xmt}}\ m'] \wedge m \sim m'))\textbf{\textit{U}}[c_{(d,r)}\ \textbf{\textit{xmt}}\ m] \end{aligned}$$

**Definition 10.9** (Assumptions about Introduction)**.**
*Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path. Predicate asmIntro*$(x, (s, d, r))$ *holds whenever* $x$ *assumes that property intro (Definition 10.8) holds.*

Message are duplicated if a two messages of the same color may occur on an output channel.

**Definition 10.10** (Duplication)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path. A message on path* $(s, d, r)$ *is duplicated if*

$$\begin{array}{l} \textit{duplicates}((s,d,r)) \equiv \\ \exists m, m' : [c_{(d,r)}\ \textbf{\textit{xmt}}\ m] \wedge \textbf{\textit{EXEF}}[c_{(d,r)}\ \textbf{\textit{xmt}}\ m'] \wedge m \sim m' \end{array}$$

**Definition 10.11** (Assumptions about no duplication)**.**
*Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path. Predicate asmDuplicate*$((s, d, r))$ *represents if the sender (the receiver) assumes that property* no duplication *holds on path* $(s, d, r)$*.*

The content of messages may change on a path if two messages have the same color ($m \sim m'$), but not the same content ($m \neq m'$):

**Definition 10.12** (Changes)**.**
*Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $M = (P, L_P, D, T)$ *be a model, and let* $(s, d, r)$ *be a path in model* $M$*. A messages is changed on path* $(s, d, r)$ *if*

$$\begin{array}{lcl} \textit{changes}((s,d,r)) & \equiv & \exists m, m'. \\ & & [c_{(s,d)}\ \textbf{\textit{xmt}}\ m] \wedge \textbf{\textit{EF}}[c_{(d,r)}\ \textbf{\textit{xmt}}\ m'] \\ & & \wedge\ m \sim m' \wedge m \neq m' \end{array}$$

**Definition 10.13** (Assumptions about Changes)**.**
*Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path. Predicate asmChanges*$(x, (s, d, r))$ *holds whenever* $x$ *assumes that property changes (Definition 10.12) holds.*

Messages are mixed on a channel if a message $m$ is send before a message $n$, but messages $m'$ (with $m \sim m'$) is received after $n'$ (with $n \sim n'$):

**Definition 10.14** (Mixes)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path. Let* $m$ *and* $n$ *be messages on channel* $c_{(s,d)}$*, and let* $m'$*,* $n'$ *be messages on channel* $c_{(d,r)}$*. Messages on path* $(s, d, r)$ *are mixed if*

$$\begin{array}{l} \textit{mixes}((s,d,r)) \equiv \\ \exists m, m', n, n' : \ [c_{(s,d)}\ \textbf{\textit{xmt}}\ m] \wedge \textbf{\textit{EXEF}}[c_{(s,d)}\ \textbf{\textit{xmt}}\ n] \\ \qquad\qquad\quad \wedge \textbf{\textit{EF}}([c_{(d,r)}\ \textbf{\textit{xmt}}\ m'] \wedge \textbf{\textit{EF}}[c_{(d,r)}\ \textbf{\textit{xmt}}\ n']) \\ \qquad\qquad\quad \wedge\ m \sim m' \wedge n \sim n' \\ \qquad\qquad\quad \wedge \textbf{\textit{E}}(\neg[c_{(d,r)}\ \textbf{\textit{xmt}}\ m'])\textbf{\textit{U}}[c_{(d,r)}\ \textbf{\textit{xmt}}\ n'] \end{array}$$

**Definition 10.15** (Assumptions about mixing)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path. Predicate asmMixes*$((s, d, r))$ *represents if the sender (the receiver) assumes that property* mixes *holds on path* $(s, d, r)$*.*

Note that losslessness is *not* a required property. As we will see in the CC concept definitions in Section 10.2.1, formal definitions of concepts relying on trusted paths can be given without requiring losslessness.

**Definition 10.16** (Assumptions about Modifications)**.**
*Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path. Predicate asmModifies*$(x, (s, d, r))$ *holds whenever* $x$ *assumes that property modifies((s,d,r)) (Definition 10.5) holds.*

Note that these assumptions and properties may change. For example, a user may assume that he is communicating with application $A$ via the keyboard for some time, while later on—after switching to a different application—he may assume that he is communicating with application $B$ now. We call this *opening*, *maintaining*, and *closing* a communication path. Opening a communication path $(s, d, r)$ means that a trusted path is established. Closing a communication path means that the path is no longer trusted.

**Definition 10.17** (Opening, Maintaining, and Closing a Communication Path)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path.*

$$
\begin{aligned}
\textit{opening}((s,d,r)) &\equiv \neg\textit{trusted}((s,d,r)) \\
&\quad \wedge \textbf{\textit{AX}}\textit{trusted}((s,d,r)) \\
\textit{maintaining}((s,d,r)) &\equiv \textit{trusted}((s,d,r)) \\
&\quad \wedge \textbf{\textit{AX}}\textit{trusted}((s,d,r)) \\
\textit{closing}((s,d,r)) &\equiv \textit{trusted}((s,d,r)) \\
&\quad \wedge \textbf{\textit{AX}}\neg\textit{trusted}((s,d,r))
\end{aligned}
$$

### 10.2.2 Definitions of CC Concepts

The core definitions from Section 10.2.1 allow to formalize relevant subclasses of Common Criteria classes FTP, FIA, FTA, FRU, and FCO.

- Class FTP: Trusted path/channels

  This class cataloguizes requirements for trusted paths within a system (TSF_ITC), and in the communication of a system with a user (FTP_TRP). Only the latter is relevant for HCI security.

> "Families in this class provide requirements for a trusted communication path between users and the TSF, and for a trusted communication channel between the TSF and other trusted IT products. Trusted paths and channels have the following general characteristics:
>
> – The communications path is constructed using internal and external communications channels (as appropriate for the component) that isolate an identified subset of TSF data and commands from the remainder of the TSF and user data.
> – Use of the communications path may be initiated by the user and/or the TSF (as appropriate for the component)
> – The communications path is capable of providing assurance that the user is communicating with the correct TSF, and that the TSF is communicating with the correct user (as appropriate for the component)" (Common Criteria Evaluation Board (CCEB), 2006, part 2, page 168)

> "FTP_TRP.1.2: The TSF shall permit [selection: the TSF, local users, remote users] to initiate communication via the trusted path."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 172)

We assume that requests to open a trusted path $((x, d, r))$ are issued by sending messages. In the following definition, we assume that predicate *reqOpen* holds when a message request the opening of a channel. If such a message send on a given path $(s', d', r')$, then a trusted path $(s, d, r)$ shall eventually be opened:

**Definition 10.18** (FTP_TRP.1.2)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model.*

*Let* $(s, d, r)$ *be the path that should become a trusted path, and let* $(s', d', r')$ *be the path for requesting opening of the trusted path. Let reqOpen*$(m)$ *be true if message* $m$ *requests opening of the trusted path. Then*

$$\textit{FTP\_TRP.1.2}((s, d, r), (s', d', r')) \equiv$$
$$\forall\, m.[c_{(s',d')}\ \textbf{\textit{xmt}}\ m] \wedge \textit{reqOpen}(m) \rightarrow \textbf{\textit{AF}}\textit{opening}((s, d, r))$$

> "FTP_TRP.1.3: The TSF shall require the use of the trusted path for [selection: initial user authentication, [assignment: other services for which trusted path is required]]."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 171)

In Section 9.2, the formal concepts *secret* and *legitimate* have been introduced (Definition 9.10). These are used in the formal definition of FTP_-TRP.1.3. In the definition of FTP_TRP.1.3, $X(m)$ indicates that $m$ is part of a service requiring a trusted path. FTP_TRP.1.3 is defined as

**Definition 10.19** (FTP_TRP.1.3)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $X$ *be a predicate indicating if a message is part of the service for which the trusted path is required. Then*

$$\textit{FTP\_TRP.1.3}(s, d, r, X) \equiv$$
$$\forall\, m.X(m) \wedge [c_{(s,d)}\ \textbf{\textit{xmt}}\ m] \rightarrow \textit{trusted}((s, d, r))$$

- Class FIA: Identification and authentication

An important aspect of trusted paths is the identification of the parties involved in the communication. The Common Critiera dedicate a whole class to this, FIA: Identification and authentication. Correctness of the actual protocols used for authentication is not the subject of this thesis. We assume the authentication protocols are correct, and give formal definitions of the Common Criteria subclasses related to authentication.

> "FIA_AFL.1.1: The TSF shall detect when [assignment: number] unsuccessful authentication attempts occur related to [assignment: list of authentication events]."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 89–90)

Authentication was succesful if a state of non-authentication is followed by a state of authentication. We assume predicate *authenticated* holds on successful authentication, and predicate *authFailed* holds if an authentication attempt failed. A counter *authAttemptsCounter* is defined as follows: If authentication was successful , then *authAttemptsCounter*$(0)$ holds in the next step. If authentication failed, then *authAttemptsCounter* is increased by 1 in the next step. If neither a successful nor an unsuccessful authentication attempt happened, the *authAttemptsCounter* stays the same.

**Definition 10.20** (Counting Authentication Attempts)**.**
*Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let authenticated hold when an authentication attempt was succesful, and let authFailed hold when an authentication attempt failed. A counter numFailedAuth*$(n)$ *for the*

*number of failed attempts is defined as*

$$
\begin{aligned}
&\mathit{authAttemptsCounter} \equiv \\
&(\mathit{authenticated} \rightarrow \\
&\quad \mathbf{AX}\mathit{numFailedAuth}(0) \\
&\quad \wedge \forall m.\mathbf{AX}(m \neq 0 \rightarrow \neg \mathit{numFailedAuth}(m))) \\
&\wedge (\mathit{authFailed} \wedge \mathit{numFailedAuth}(n) \rightarrow \\
&\quad \mathbf{AX}\mathit{numFailedAuth}(n+1) \\
&\quad \wedge \forall m.\mathbf{AX}(m \neq n+1 \rightarrow \neg \mathit{numFailedAuth}(m))) \\
&\wedge (\neg \mathit{authFailed} \wedge \neg \mathit{authenticated} \wedge \mathit{numFailedAuth}(n) \rightarrow \\
&\quad \mathbf{AX}\mathit{numFailedAuth}(n) \\
&\quad \wedge \forall m.\mathbf{AX}(m \neq n \rightarrow \neg \mathit{numFailedAuth}(m)))
\end{aligned}
$$

> "FIA_AFL.1.2: When the defined number of unsuccessful authentication attempts has been met or surpassed, the TSF shall [assignment: list of actions]."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 90)

**Definition 10.21** (FTP_AFL.1.2)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $n$ *be the maximal number of authentication attempts. Let maxAuthExceededAction hold when the actions assigned to an excess of the maximum number of authentication attempts are executed. Then*

$$
\begin{aligned}
&\mathit{FTP\_AFL.1.2}(n) \equiv \\
&\mathit{authAttemptsCounter} \wedge \\
&(\mathit{numFailedAuth}(n) \rightarrow \mathbf{AX}\mathit{maxAuthExceededAction})
\end{aligned}
$$

> "FIA_ATD.1.1: The TSF shall maintain the following list of security attributes belonging to individual users: [assignment: list of security attributes]."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 91)

User security attributes are defined by predicates. The actual predicates depend on the application. In the context of confidentiality requirement we defined predicate *secret* and *legitimate* (Definition 9.10).

From the type of authentication mechanisms defined in FIA_UAU, FIA_-UAU.1/FIA_UID.1 and FIA_UAU.2/FIA_UID.2 deal with operations possible before authentication:

> "FIA_UAU.1 Timing of authentication, allows a user to perform certain actions prior to the authentication of the user's identity.

> "FIA_UAU.2 User authentication before any action, requires that users authenticate themselves before any action will be allowed by the TSF."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 94)

We define legitimate actions via *legitimate*. If a message may be send prior authentication, then it is always legitimate to send the message:

**Definition 10.22** (FIA_UAU.1)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $d$ *be the device, and* $l$ *the set of set of messages allowed to be communicated before authentication. Then*

$$\textit{FIA\_UAU.1}(d, l) \equiv$$
$$\boldsymbol{AG} \forall\, s, m.m \in l \rightarrow \textit{legitimate}(s, d, a, m)$$

If the user must be authenticated before sending any message, then sending a message is legitimate only if the path is authenticated:

**Definition 10.23** (FIA_UAU.2)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $d$ *be the device. Then*

$$\textit{FIA\_UAU.2}(d) \equiv$$
$$\forall\, m.\textit{legitimate}(u, d, a, m) \rightarrow \textit{authenticated}((u, d, a))$$

FIA_UAU.3 requires unforgeable authentication, FIA_UAU.4 single-use authentication, FIA_UAU.5, multiple authentication mechanisms, and FIA_UAU.7 defines requirements for feedback in the authentication process. These are not subject of this thesis, because we treat the authentication process as a "black box"; we do not model the actual method used for authentication.

> "FIA_UAU.6 Re-authenticating, requires the ability to specify events for which the user needs to be re-authenticated."

If the system is in a state requiring re-authentication, then the path between user and application will not be authenticated in the next step:

**Definition 10.24** (FIA_UAU.6)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $d$ *be an input device and let* $t$ *be the set of states for which re-authentication is required.*

$$\textit{FIA\_UAU.6}(d, t) \equiv$$
$$\forall\, x.\textit{state}(x) \wedge x \in t \rightarrow \boldsymbol{AX} \neg \textit{authenticated}(u, d, a)$$

User-subject binding is defined in FIA_USB as:

> "FIA_USB.1.1: The TSF shall associate the appropriate user security attributes with subjects acting on behalf of that user."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 101)

This is achieved by associating users with paths. This is achieved by Definition 10.4 (Authentication). In the definition of *legitimate*, the security attributes are defined on combinations of users, paths, and receivers.

- Class FTA: TOE access

  Class FTA: TOE access deals with session management. In the context of HCI, sessions are closely related to the access to I/O resources. Therefore, the relevant parts of this class are similar to the relevant parts of the classes dealing with I/O resource allocation. Class FTA_SSL (Session locking) addresses the question of maintaining and re-establishing a trusted session. From the subclasses of class FTA (TOE Access), FTA_MCS (Limitation on multiple concurrent sessions (FTA_MCS) is relevant for HCI security:

  > "FTA_MCS.1.1: The TSF shall restrict the maximum number of concurrent sessions that belong to the same user.
  >
  > "FTA_MCS.1.2: The TSF shall enforce, by default, a limit of [assignment: default number] sessions per user."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 161)

This is achieved by enforcing a maximal number of authenticated paths associated with a user at all times:

**Definition 10.25** (Set of trusted paths)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. The set of trusted paths (paths) is defined as*

$$\textit{paths} \equiv \{(u, d, a) \mid \textit{trusted}((u, d, a)) \textit{ for some } d\}$$

We formalize FTA_MCS.1:

**Definition 10.26** (FTA_MCS.1)**.** *Let* $m$ *be the maximal number of sessions per user. FTA_MCS.1 is satisfied if*

$$\textit{FTA\_MCS.1}(m) \equiv |\textit{paths}| \leq m$$

FTA_SSL:

> "This family defines requirements for the TSF to provide the capability for TSF- initiated and user-initiated locking and unlocking of interactive sessions."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 162)

If a device (path) is locked, all incoming messages will be discarded, i.e. all input arriving on a locked device is discarded; also, messages which arrived before the device was locked will not be send while the device is locked. The formal specification of *Path Locking* consists of two conjuncts. The first conjunction defines that if a message $m$ is send while the path is locked, then no message $m'$ of matching color ($m \sim m'$) will ever be received, even if the path is unlocked later on. The second conjunct defines that no message is received while the path is locked.

**Definition 10.27** (Path Locking)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Locking of a path* $(s, d, r)$ *is defined as*

$$\begin{aligned} \mathit{locked}((s,d,r)) \equiv \;& \forall\, m, m', m''. \\ & [c_{(s,d)}\ \textbf{\textit{xmt}}\ m] \rightarrow \\ & \quad \neg(\textbf{\textit{EF}}[c_{(d,r)}\ \textbf{\textit{xmt}}\ m'] \wedge m \sim m') \\ & \wedge \neg[c_{(d,r)}\ \textbf{\textit{xmt}}\ m''] \end{aligned}$$

If a path is completely locked, no interaction is possible. The definition of FTA_SSL.1.1 however requires that operations for unlocking the path are still possible. The formal definition of *Partial Path Locking* is very similar to the definition of *Path Locking*. The only difference is that messages which satisfy property *authMesg* may be communicated on partially locked paths:

**Definition 10.28** (Partial Path Locking)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let authMsg*$(m)$ *hold whenever a message* $m$ *is part of the authentication protocol. Then partial locking is defined as*

$$\begin{aligned} \mathit{partLocked}((s,d,r)) \equiv \;& \forall\, m, m', m''. \\ & [c_{(s,d)}\ \textbf{\textit{xmt}}\ m] \wedge \neg \mathit{authMsg}(m) \rightarrow \\ & \quad \neg(\textbf{\textit{EF}}[c_{(d,r)}\ \textbf{\textit{xmt}}\ m'] \wedge m \sim m') \\ & \wedge \neg[c_{(d,r)}\ \textbf{\textit{xmt}}\ m''] \vee \mathit{authMsg}(m'') \end{aligned}$$

> "FTA_SSL.1.1: The TSF shall lock an interactive session after [assignment: time interval of user inactivity] by:
> - clearing or overwriting display devices, making the current contents unreadable;
> - disabling any activity of the user's data access/display devices other than unlocking the session."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 162)

In our framework, time is represented as time steps. The property of the user not making an input for $n$-time steps is formalized in a way very similar to the formalization of the counter of failed authentication attempts (Definition 10.20). When the user sends a message ($[c_{(u,d,)}$ **xmt** $m]$), the counter *noInput* is set to zero in the next step. If the user does not send a message, then *noInput* is increased by one in the next step:

**Definition 10.29** (Time without User Input)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. The time without user input on a path* $(u, d, r)$ *is defined as*

$$
\begin{aligned}
&\mathit{noUserInputCounter}((u,d,r)) \equiv \\
&((\exists m.[c_{(u,d)}\ \textbf{\textit{xmt}}\ m]) \rightarrow \\
&\qquad \textbf{\textit{AX}}\mathit{noInput}((u,d,r),0) \\
&\qquad \wedge \forall p.\textbf{\textit{AX}}(p \neq 0 \rightarrow \neg \mathit{noInput}((u,d,r),p))) \\
&\wedge \forall m'.(\neg[c_{(u,d)}\ \textbf{\textit{xmt}}\ m'] \wedge \mathit{noInput}((u,d,r),n) \rightarrow \\
&\qquad \textbf{\textit{AX}}\mathit{noInput}((u,d,r),n+1)) \\
&\qquad \wedge \forall p.\textbf{\textit{AX}}(p \neq n+1 \rightarrow \neg \mathit{noInput}((u,d,r),p)))
\end{aligned}
$$

With the definition of *noInput* it is possible to FTA_SSL.1.1. In the formalization of FTA_SSL.1.1, we assume that two paths should be locked after no user activity: The path from the user to the application via an input device $d$, and the path from the application to the user via an output device $d'$.

**Definition 10.30.**
*Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(u, d, a)$ *be the input path and* $(a, d', u)$ *be the output path. Let* $n$ *be the time after which the paths shall be locked in case of no user input. Then*

$$
\begin{aligned}
&\mathit{FTA\_SSL.1.1}(d, d', n) \equiv \\
&\mathit{noInput}((u,d,a),m) \wedge n \leq m \rightarrow \\
&\quad \mathit{partLocked}((u,d,a)) \wedge \mathit{partLocked}((u,d',a))
\end{aligned}
$$

In our definitions of *locked* and *partLocked*, no messages are output on a blocked device. Therefore, explicitly cleaning the screen is not necessary. In Section 14.2 we show how a realistic screen component is modeled and how it affects system security.

FTA_SSL.2 and FTA_SSL.3 require that locking can be initiated by the user (FTA_SSL.2) and the application (FTA_SSL.3). We assume there is a predicate *lockIt* which holds for messages requesting locking. When a message satisfying *lockIt* is send, then the path from the user to the application ($(u, d, a)$), and the path from the application to the user ($(a, d', u)$) shall be locked in the next step:

**Definition 10.31** (FTA_SSL.2)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(u, d, a)$ *be the input path and* $(a, d', u)$ *be the output path. Let predicate lockIt*$(m)$ *indicate if* $m$ *is a message requesting locking. Then*

$$\begin{aligned}&\textit{FTA\_SSL.2}(d, d') \equiv \\ &\forall m.[c_{(u,d)}\ \textbf{\textit{xmt}}\ m] \wedge \textit{lockIt}(m) \rightarrow \\ &\quad \textbf{\textit{AX}}(\textit{partLocked}((u, d, a)) \wedge \textit{partLocked}((a', d, u')))\end{aligned}$$

> "FTA_TAB.1.1 Before establishing a user session, the TSF shall display an advisory warning message regarding unauthorised use of the TOE."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 165)

We assume *banner*$(m)$ holds if $m$ is a message showing the advisory message. If a trusted channel is opened, an advisory message should be shown in the next step:

**Definition 10.32** (FTA_TAB.1.1)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let banner*$(m)$ *be true if message* $m$ *is a banner. Let* $d$ *be the device where the banner shall be shown.*

$$\begin{aligned}&\textit{FTA\_TAB.1.1}(d) \equiv \\ &\exists m.\textit{banner}(m) \wedge \\ &\textit{opening}((a, d, u)) \rightarrow \textbf{\textit{AX}}[c_{(d,u)}\ \textbf{\textit{xmt}}\ m])\end{aligned}$$

> "FTA_TAH.1.1 Upon successful session establishment, the TSF shall display the [selection: date, time, method, location] of the last successful session establishment to the user."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 166)

The formal definition of FAT_TAH.1.1 is very similar to the definition of FTA_TAB.1.1. We assume *est*$(m)$ holds if $m$ is a message showing the information about last successful session establishment. If a trusted channel is opened, an information message should be shown in the next step:

**Definition 10.33** (FTA_TAH.1.1)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let est*$(m)$ *be true if message* $m$ *displays the relevant information. Let* $d$ *be the device where the information shall be shown.*

$$\begin{aligned}&\textit{FTA\_TAH.1.1}(d) \equiv \\ &\exists m.\textit{est}(m) \wedge \\ &\textit{opening}((a, d, u)) \rightarrow \textbf{\textit{AX}}[c_{(d,u)}\ \textbf{\textit{xmt}}\ m])\end{aligned}$$

> "FTA_TAH.1.2 Upon successful session establishment, the TSF shall display the [selection: date, time, method, location] of the last unsuccessful attempt to session establishment and the number of unsuccessful attempts since the last successful session establishment."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 166)

The formal definition of FTA_TAH.1.2 is very similar to FTA_TAB1.1 and FTA_TAH.1.1. In the definition of FTA_TAB1.1, *banner*$(m)$ is true for messages showing a banner. In the definition of FTA_TAH1.1, *est*$(m)$ is true for messages showing the relevant information about session establishment. In the definition of FTA_TAH1.2, *failedAttempts*$(m)$ is true for messages showing information about the number of failed authentication attempts.

**Definition 10.34** (FTA_TAH.1.2)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let failedAttempts*$(m)$ *be true if message* $m$ *displays the relevant information about failed authentication attempts. Let* $d$ *be the device where the information shall be shown.*

$$\begin{aligned}&\textit{FTA\_TAH.1.2}(d) \equiv\\ &\exists m.\textit{failedAttempts}(m) \wedge\\ &\textit{opening}((a, d, u)) \rightarrow \mathbf{AX}[c_{(d,u)}\ \textbf{\textit{xmt}}\ m])\end{aligned}$$

> "FTA_TAH.1.3 The TSF shall not erase the access history information from the user interface without giving the user an opportunity to review the information."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 166)

Since we have a integrated view on HCI security, we strengthen FTA_TAH.1.3 to the requirement that the user has recognized the system state. If a message $m$ with the information about failed authentication attempts ($mi failedAttempts(m)$ holds) is send on the path, then a message $m'$ of the same color ($m \sim m'$) will hold until the user's assumption about failed attempts is identical to the information in message $m$ ($mi asmFailedAttempts(u, m)$) holds.

**Definition 10.35** (FTA_TAH.1.3)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let failedAttempts*$(m)$ *be true if message* $m$ *displays the relevant information. Let asmFailedAttempts*$(m)$ *hold if the user assumes that the access history information is identical to the information in message* $m$*. Let* $d$ *be the device where the information shall be shown.*

$$\begin{aligned}&\textit{FTA\_TAH.1.3}(d) \equiv\\ &\exists m.\textit{failedAttempts}(m) \wedge ([c_{(a,d)}\ \textbf{\textit{xmt}}\ m]) \rightarrow\\ &\mathbf{AXA}(([c_{(d,u)}\ \textbf{\textit{xmt}}\ m']) \wedge m \sim m')\mathbf{U}\textit{asmFailedAttempts}(u, m)\end{aligned}$$

TOE session establishment (FTA_TSE) (Common Criteria Evaluation Board (CCEB), 2006, part 2, page 167) is not specific to user interface security.

## 10.3 Privacy and Confidentiality (FDP, FPR)

### 10.3.1 Overview

Privacy and confidentiality requirements are covered in two classes of the Common Criteria. Class FDP defines user data protection requirements. Class FPR defines privacy requirements. Class FDP (User data protection) is described as follows:

> "This class contains families specifying requirements related to protecting user data. FDP: User data protection is split into four groups of families (listed below) that address user data within a TOE, during import, export, and storage as well as security attributes directly related to user data."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 54)

Only some of the sub-classes of FDP are relevant for HCI security. Classes FDP_ACC and FDP_IFC define access and information flow control policies. FDP_RIP deals with secure information deletion. FDP_ROL defines requirements for returning to a previous state. The other classes of this sub-set are not relevant to human-computer interaction: FDP_ITT is concerned with data transfer within a system. This is not relevant for our work, because we only model data flow between the user and the application. Also, FDP_SDI is not relevant because it defines requirements for stored data integrity. FDP_ACF and FDP_IFF provide a list of requirement definitions for fine-grained information control policies. Since this is not directly relevant to HCI security, we do not formalize FDP_ETC,which governs the export of data outside the TSF. FDP_DAU defines requirements for guaranteeing authenticity of data. FDP_UCT defines confidentiality requirements and class FDP_UIT integrity requirements

### 10.3.2 Core Definitions

In the Common Criteria, privacy is understood as Anonymity (non-disclosure of user's identity), Pseudonymity (non-disclosure of user's identity while still being accountable), Unlinkability (multiple uses of resources can not be linked by third parties), and Unobservability (third parties can not observe if a user is using a service).(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 118–125) The Common Criteria definition of privacy is concerned with protecting data from disclosure, like in the following example:

> "In FPR_ANO.2.2 the PP/ST author should identify the list of services which are subject to the anonymity requirement, for example,

> 'the accessing of job descriptions'". (Common Criteria Evaluation Board (CCEB), 2006, part 2, page 268)

In the context of HCI interaction, we are only interested in the privacy of the communication channels between the user and the application. Therefore, Common Criteria class FPR is not formalized here.

For HCI privacy, all information send via a path $(s, d, r)$ is received by the legitimate receiver only, i.e. if somebody gets a delivery, it must come from the correct device and sender. If a message $m$ send on the path ($[c_{(s,d)}$ **xmt** $m]$) and a message of the same color is eventually received somewhere else (**EF**$[c_{(d',r')}$ **xmt** $m'] \wedge m \sim m'$), then it should be received on the same path, i.e. $d = d' \wedge r = r'$:

**Definition 10.36** (Private)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model, and let* $(s, d, r)$ *be a path in model* $M$*. The path is* private *if*

$$\begin{aligned}&\textit{private}((s,d,r)) \equiv \\ &[c_{(s,d)}\ \textbf{\textit{xmt}}\ m] \wedge \textbf{\textit{EF}}[c_{(d',r')}\ \textbf{\textit{xmt}}\ m'] \wedge m \sim m' \rightarrow \\ &d = d' \wedge r = r'\end{aligned}$$

**Definition 10.37** (Assumptions about privacy)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(s, d, r)$ *be a path. Predicate asmPrivate*$(x, (s, d, r))$ *holds whenever* $x$ *assumes that property private*$((s, d, r))$ *(Definition 10.36) holds.*

### 10.3.3 Definitions of CC Concepts

> "FDP_IFC.1.1 The TSF shall enforce the [assignment: information flow control SFP] on [assignment: list of subjects, information, and operations that cause controlled information to flow to and from controlled subjects covered by the SFP]."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 66)

We use *secret* and *legitimate* (Definition 9.10) in the definition of FDP_IFC.1.1. FDP_IFC.1.1 is satisfied if all messages $m$ declared secret if send from $s$ to $r$ via $d$ (*secret*$(s, d, r, m)$) are send only if sending is legitimate (*legitimate*$(s, d, r, m)$):

**Definition 10.38** (FDP_IFC.1.1)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model.*

$$\begin{aligned}&\textit{FDP\_IFC.2.1} \equiv \\ &\forall s, d, r, m.[c_{(s,d)}\ \textbf{\textit{xmt}}\ m] \wedge \textit{secret}(s,d,r,m) \rightarrow \\ &\qquad\qquad \textit{legitimate}(s,d,r,m)\end{aligned}$$

> "FDP_IFC.2.2 The TSF shall ensure that all operations that cause any information in the TSC to flow to and from any subject in the TSC are covered by an information flow control SFP."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 66)

Message flow is covered by an information control flow policy if the message is declared secret. In order to satisfy FDP_IFC.2.2 all messages which may be potentially send on any device from the user to the application or from the application to the user must be declared secret:

**Definition 10.39** (FDP_IFC.2.2)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model.*

$$
\begin{aligned}
&\textit{FDP\_IFC.2.2} \equiv \\
&\forall\, d, m, r. \\
&(d \in I \rightarrow \textit{secret}(u, d, r, m)) \\
&\wedge\, (d \in O \rightarrow \textit{secret}(a, d, r, m))
\end{aligned}
$$

FDP_RIP (Residual information protection) deals with data deletion: "This family addresses the need to ensure that deleted information is no longer accessible, and that newly created objects do not contain information that should not be accessible."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 77).

The only requirement in this class is FDP_RIP.1.1:

> "FDP_RIP.1.1 The TSF shall ensure that any previous information content of a resource is made unavailable upon the [selection: allocation of the resource to, deallocation of the resource from] the following objects: [assignment: list of objects]."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 77).

In respect to the user interface, we have to make sure that data coming from and going to the I/O devices is securely deleted. For example, sensitive data should be deleted from the keyboard buffer before a different application gets access to it, and from the screen before a different user gets access to it. When output devices like the screen are updated asynchronously, it may happen that logically deleted information is still present on the screen. For human-computer interaction, this is identical to the locking requirement of FTA_SSL.1 and FTA_SSL.2 for screen and keyboard. See Definitions 10.27 to 10.31.

FDP_IFF (Information Flow control functions) requires a fine grained information flow policy including hierarchical security attributes. In this work, we are content with security attributes defined by *legitimate*.

We do not formalize classes FDP_ITC (Import from outside TSF control), FDP_ITT (Internal TOE transfer), FDP_UCT (Inter-TSF user data confidentiality

transfer protection), and FDP_UIT (Inter-TSF user data integrity transfer protection), because we only consider data flow between the user and the application, not in between applications and systems.

Rollback operations (FDP_ROL) allow to return to previous states of the system.

> "FDP_ROL.1.1 The TSF shall enforce [assignment: access control SFP(s) and/or information flow control SFP(s)] to permit the rollback of the [assignment: list of operations] on the [assignment: list of objects]."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 79–80)

In our system model, states of the system are represented as states of IOLTS. Predicate *state*$(x)$ indicates if a system component is in state $x$ (Definition 4.9). We assume there are two states $x$ and $y$ and a message $m$. The application shall rollback from state $y$ to state $x$ if message $m$ is send. For this, two conditions have to be satisfied. First, the application must be in state $x$ and will eventually be in state $y$ (*state*$(x) \wedge$ **EF***state*$(y)$). Second, whenever the application is in state $y$ in the future, and message $m$ is send, then the application will be in state $x$ again, eventually, and until the application is in state $x$ again, it will never be in a fatal state (**AG**(*state*$(y) \wedge [c_{(u,d)}$ **xmt** $m] \rightarrow$ (**A**($\neg$*fatal*)**U***state*$(x)$))).

**Definition 10.40** (FDP_ROL.1.1)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $x$ *and* $y$ *be states, and let* $(u, d, a)$ *be a path from the user to the application. Let* $m$ *be a message which, if send on path* $(u, d, a)$*, initiates rollback from state* $y$ *to state* $x$*. Then*

$$
\begin{array}{l}
\textit{FDP\_ROL.1.1}(x, y, (u, d, a), m) \equiv \\
\textit{state}(x) \wedge \textbf{\textit{EF}}\textit{state}(y) \rightarrow \\
\quad \textbf{\textit{AG}}(\textit{state}(y) \wedge [c_{(u,d)}\ \textbf{\textit{xmt}}\ m] \rightarrow \\
\qquad (\textbf{\textit{A}}(\neg \textit{fatal})\textbf{\textit{U}}\textit{state}(x)))
\end{array}
$$

> "FDP_ROL.1.2 The TSF shall permit operations to be rolled back within the [assignment: boundary limit to which rollback may be performed]."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 80)

For this, we define a relation *rollbackPossible*$(x, y)$, indicates that rollback from state $y$ to state $x$ is possible. Furthermore, let *rollbackCommand* be a function from a tuple of states to a message, indicating that the message initiates a rollback from the second element of the tuple to the first element of the tuple. Then FDP_ROL.1.2 is satisfied if FDP_ROL.1.1 holds for all $x$ and $y$ for which rollback is possible with rollback command *rollbackCommand*$(x, y)$:

**Definition 10.41** (FDP_ROL.1.2)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Let* $(u, d, a)$ *be a path from the user to the application. Let rollbackPossible be a relation with two states as arguments, and let rollbackCommand be a mapping from tuples of states to messages. Then*

$$\begin{aligned}&\textit{FDP\_ROL.1.2}((u, d, a), \textit{rollbackPossible}, \textit{rollbackCommand}) \equiv \\ &\forall x, y, m.\textit{rollbackPossible}(x, y) \rightarrow \\ &\quad \textit{FDP\_ROL.1.1}(x, y, (u, d, a), \textit{rollbackCommand}(x, y))\end{aligned}$$

FDP_DAU defines requirements for guaranteeing authenticity of data:

> "Data authentication permits an entity to accept responsibility for the authenticity of information (e.g., by digitally signing it)."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 61)

Digital signature methods are not subjects of HCI security, but guaranteeing that the user knows what he is signing is. When the user takes responsibility for some data (e.g. by signing it), the data stored in the system should correspond to the user's opinion about the data. We discuss this in the context of methods for guaranteeing data integrity in Chapter 11. Sub-concepts FDP_UCT and FDP_UIT require a definition of data privacy:

> "FDP_UCT.1.1 The TSF shall enforce the [assignment: access control SFP(s) and/or information flow control SFP(s)] to be able to [selection: transmit, receive] objects in a manner protected from unauthorised disclosure."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 83)

> "FDP_UIT.1.1 The TSF shall enforce the [assignment: access control SFP(s) and/or information flow control SFP(s)] to be able to [selection: transmit, receive] user data in a manner protected from [selection: modification, deletion, insertion, replay] errors."(Common Criteria Evaluation Board (CCEB), 2006, part 2, page 85)

This is identical to our definition of the definitions of trusted paths (Definition 10.1) in Section 10.2.1.

# Chapter 11

# Confidentiality–Integrity–Availability

## 11.1 Definitions

In the last chapter, the Common Criteria concepts relevant to HCI security have been formalized. The Common Criteria are a collection of security requirements. Developers and evaluators of secure systems have to decide which parts of the Common Criteria are relevant for their projects, and which parts are not. There is no guarantee that the developer/evaluator has chosen the right elements of the Common Criteria, and that no relevant aspects have been omitted. Furthermore, the Common Criteria provide a criteria catalog for computer systems only. It does not define user behavior requirements.

The Common Criteria are a "bottom up" approach to computer security. In this chapter, we accompany the formal definitions of Common Criteria concepts by a "top down" approach. We start by formalizing the well-established definition of computer security as *Confidentiality*, *Integrity*, and *Availability* (also known as the *CIA-model*) for HCI security. In Section 11.2, we show that the core concepts developed for the formal definition of Common Criteria requirements can also be used to formalize the generic CIA requirements. The CIA-based formal requirement definitions for computer systems are accompanied by requirement definitions for user behavior. This allows to prove that suitable combinations of application and user behavior components guarantee that no security breaches occur. The joined approach of "bottom up" Common Criteria concepts on the one hand and "top down" CIA concepts on the other hand allows for the pervasive specification of secure human-computer interaction.

In the field of information security, the basic security threats are identified as *Data Leaking*, *Data Manipulation*, and *Program Manipulation* (see e.g. Clark and

Wilson (1987); ITS; Dierstein (2004)). These are countered by the core security requirements, usually abbreviated as *CIA*:

| | |
|---|---|
| **Confidentiality** | Information is available to authorized parties only. |
| **Integrity** | Neither the system nor services provided by and data processed by the system can be manipulated. Third parties accessing the system can not assume the identity of a legitimate user. |
| **Availability** | Accessibility of services and data is guaranteed. |

We adapt these concepts to user interface security by restricting these definitions to the aspects involving the user interface and human-computer interaction. For Confidentiality, this means that eavesdropping on the input/output facilities must not be possible. Integrity of the user interface is guaranteed if manipulation of the user interface is not possible, i.e. if the user's assumptions about the state of the application, gained by observing and manipulating the application via the user interface, corresponds to the actual state of the application. Availability of the user interface means that an attacker can not get the user interface into a state where the full functionality is no longer accessible.

| | |
|---|---|
| **Confidentiality** | A third party can not gain information from observing human-computer interaction. |
| **Integrity** | Whenever the user issues a command, all relevant information, most notably the state of the program and the data processed, is shown on the screen correctly. |
| **Availability** | The functionality provided by the user interface is always accessible. |

We bridge the gap between the abstract concept definitions of CIA and concrete security requirements specification for applications by breaking down CIA concepts into sub-concepts. In the definition of sub-concepts, we make use of the core concepts developed for the formalization of Common Criteria requirements. This way, we create a common base for security specifications based on CIA, and security specifications based on CC. This last step serves two purposes: It shows that the concepts of the Common Criteria are suitable for a pervasive specification of HCI security, and it breaks down generic concepts of CIA into suitable requirement specifications of applications and users.

- Confidentiality

  Confidentiality is given if no secret information is leaked, i.e. whenever secret information is send, it either reaches a legitimate receiver or no receiver at all:

  **Definition 11.1** (Confidentiality)**.**
  *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Channel* $(s, d, r)$ *is*

confidential *if*

$$\begin{array}{l}\textit{Confidentiality} \equiv \\ \forall\, s, d, r, d', r', m, m'. \\ [c_{(s,d)}\ \textbf{\textit{xmt}}\ m] \wedge \textbf{\textit{EF}}[c_{(d',r')}\ \textbf{\textit{xmt}}\ m'] \wedge m \sim m' \wedge \textit{secret}(s, d', r', m') \\ \rightarrow \\ \textit{legitimate}(s, d', r', m')\end{array}$$

- Integrity

  HCI integrity is defined as a relation between the state of the application and the user's representation of the state of the application. The user's opinion of the state should correspond to the actual state whenever the user makes critical decisions. We formalize this concept by defining attributes $a_0, \ldots, a_n$ representing the relevant aspects of the system configuration and attributes $u_0, \ldots, u_n$ representing the user's opinion about the system configuration. Furthermore, we assume that predicate *appCritical* holds whenever the application is in a critical state.

  **Definition 11.2** (Integrity)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model.*

  *Let* $a_0, \ldots, a_n$ *be the attributes representing the configuration of the application, and let* $u_0, \ldots, u_n$ *be the user's representation of these attributes. Let appCritical hold whenever the application is in a critical state. Integrity is guaranteed if*

  $$\begin{array}{l}\textit{Integrity} \equiv \\ \textit{appCritical} \rightarrow ((a_0 \leftrightarrow u_0) \wedge (a_1 \leftrightarrow u_1) \wedge \ldots \wedge (a_n \leftrightarrow u_n))\end{array}$$

- Availability

  Availability is commonly defined as reachability of desirable states and avoidability of undesirable states. Transfered to user interface security, this definition is not sufficient. It should not only be possible for some user to reach desirable states and avoid undesirable states. For a given formal user model, desirable states should always be reached and undesirable states should never be reached. This leads to the following definition:

  **Definition 11.3** (Availability)**.** *Let system*$(U, A, I, O, M, u, a, \lambda)$ *be a system model. Availability is guaranteed if*

  $$\textit{Availability} \equiv \textbf{\textit{AG}}(\neg \textit{fatal} \rightarrow \textbf{\textit{AF}}\textit{success})$$

## 11.2 Defining Confidentiality by CC Sub-Concepts

In Chapter 10, Common Criteria concepts relevant to HCI have been formalized. In Section 11.1 we formalized the generic security concepts of Confidentiality, Integrity, and Availability. In this chapter, we show how Confidentiality is decomposed in sub-concepts using core concepts from the Common Criteria. In Chapters 14 to 16, this approach is used to show that the email client developed in the Verisoft project satisfies Confidentiality, Integrity, and Availability.

Confidentiality means that information is available to authorized parties only. The definition of a confidential channels is based on the definition of trusted paths (Definition 10.1). In Section 10.2, the concept of trusted paths is one of the core concepts for the definition of various Common Criteria concepts. We show that confidentiality as defined in Definition 11.1 is guaranteed if the user sends confidential messages only if he is communicating via a confidential channel with a legitimate recipient. We call this the *Confidentiality Condition* (*ConfCond*).

**Definition 11.4** (Confidentiality Condition)**.**
*Let $S = \mathit{system}(U, A, I, O, M, u, a, \lambda)$ be a system model. S satisfies the confidentiality condition if*

$$M, \lambda \models \mathit{ConfCond}$$

*with*

$$\mathit{ConfCond} \equiv \forall\, s, d, r, m.\quad [c_{(s,d)}\ \textbf{\textit{xmt}}\ m] \wedge \mathit{secret}(s, d, r, m) \rightarrow \mathit{legitimate}(s, d, r, m) \wedge \mathit{trusted}((s, d, r))$$

We need two lemmas to show that *Confidentiality* is implied by *ConfCond*. The first lemma asserts that if a channel sends a message, and this message is received by somebody, and the channel does not leak, then the message is received by the intended recipient:

**Lemma 11.1.**

$$[c_{(s,d)}\ \textbf{\textit{xmt}}\ m] \wedge \textbf{\textit{EF}}[c_{(d',r')}\ \textbf{\textit{xmt}}\ m'] \wedge m \sim m' \wedge \neg \mathit{leaks}((s, d, r)) \rightarrow r = r' \wedge d = d'$$

**Proof**

Insert definition of *leak* (Definition 10.5 in Section 10.2.1):

$$[c_{(s,d)}\ \textbf{xmt}\ m] \wedge \textbf{EF}[c_{(d',r')}\ \textbf{xmt}\ m'] \wedge m \sim m' \wedge (\neg \exists\, m, m', d', r'.\quad [c_{(s,d)}\ \textbf{xmt}\ m] \wedge \textbf{EF}[c_{(d',r')}\ \textbf{xmt}\ m'] \wedge m \sim m' \wedge (d \neq d' \vee r \neq r')) \rightarrow r = r' \wedge d = d'$$

Moving quantifiers to begin of formula, renaming bound variables and, applying De Morgan's laws:

$$\begin{aligned}
&\forall\, m'', m''', d'', r''.\\
&[c_{(s,d)}\ \mathbf{xmt}\ m] \wedge \mathbf{EF}[c_{(d',r')}\ \mathbf{xmt}\ m'] \wedge m \sim m'\\
&\wedge\, (\neg[c_{(s,d)}\ \mathbf{xmt}\ m''] \vee \neg\mathbf{EF}[c_{(d'',r'')}\ \mathbf{xmt}\ m''']\\
&\quad \vee\, m'' \not\sim m''' \vee (d = d'' \wedge r = r''))\\
&\rightarrow r = r' \wedge d = d'
\end{aligned}$$

Since $m''$, $m'''$, $d''$, and $r''$ are universally quantified, we can set $m'' = m$, $d'' = d'$, $r'' = r$, $m''' = m'$:

$$\begin{aligned}
&[c_{(s,d)}\ \mathbf{xmt}\ m] \wedge \mathbf{EF}[c_{(d',r')}\ \mathbf{xmt}\ m'] \wedge m \sim m'\\
&\wedge\, (\neg[c_{(s,d)}\ \mathbf{xmt}\ m] \vee \neg\mathbf{EF}[c_{(d',r')}\ \mathbf{xmt}\ m']\\
&\quad \vee\, m \not\sim m' \vee (d = d' \wedge r = r'))\\
&\rightarrow r = r' \wedge d = d'
\end{aligned}$$

□

The second lemma asserts that if the channel does not change messages, than every message send will be received unchanged:

**Lemma 11.2.**

$$\begin{aligned}
&[c_{(s,d)}\ \textbf{\textit{xmt}}\ m] \wedge \textbf{\textit{EF}}[c_{(d,r)}\ \textbf{\textit{xmt}}\ m'] \wedge m \sim m'\\
&\wedge \neg \textit{changes}((s,d,r)) \rightarrow m = m'
\end{aligned}$$

**Proof**

Insert definition of *changes* (Definition 10.12 in Section 10.2.1):

$$\begin{aligned}
&[c_{(s,d)}\ \mathbf{xmt}\ m] \wedge \mathbf{EF}[c_{(d,r)}\ \mathbf{xmt}\ m'] \wedge m \sim m'\\
&\wedge \neg(\exists\, m, m'.[c_{(s,d)}\ \mathbf{xmt}\ m] \wedge \mathbf{EF}[c_{(d,r)}\ \mathbf{xmt}\ m']\\
&\qquad\qquad \wedge\, m \sim m' \wedge m \neq m')\\
&\rightarrow m = m'
\end{aligned}$$

Moving quantifiers to begin of formula, applying De Morgan's laws:

$$\begin{aligned}
&\forall\, m'', m'''.\\
&[c_{(s,d)}\ \mathbf{xmt}\ m] \wedge \mathbf{EF}[c_{(d,r)}\ \mathbf{xmt}\ m'] \wedge m \sim m'\\
&\wedge\, (\neg[c_{(s,d)}\ \mathbf{xmt}\ m''] \vee \neg\mathbf{EF}[c_{(d,r)}\ \mathbf{xmt}\ m''']\\
&\quad \vee\, m'' \not\sim m''' \vee m'' = m''')\\
&\rightarrow m = m'
\end{aligned}$$

Since $m''$ and $m'''$ are universally quantified, we can set $m'' = m$, $m''' = m'$:

$$
\begin{array}{l}
[c_{(s,d)}\ \mathbf{xmt}\ m] \wedge \mathbf{EF}[c_{(d,r)}\ \mathbf{xmt}\ m'] \wedge m \sim m' \\
\wedge\ (\neg[c_{(s,d)}\ \mathbf{xmt}\ m] \vee \neg\mathbf{EF}[c_{(d,r)}\ \mathbf{xmt}\ m'] \\
\quad \vee\ m \not\sim m' \vee m = m') \\
\rightarrow m = m'
\end{array}
$$

□

Now we can show that the confidentiality condition guarantees confidentiality:

**Theorem 11.1** (Confidentiality Condition guarantees Confidentiality)**.**

*ConfCond* $\rightarrow$ *Confidentiality*

**Proof**

Insert definitions:

$$
\begin{array}{l}
\forall\, s, d, r, m. \\
[c_{(s,d)}\ \mathbf{xmt}\ m] \wedge \textit{secret}(s, d, r, m) \rightarrow \\
\textit{legitimate}(s, d, r, m) \wedge \textit{trusted}((s, d, r)) \rightarrow \\
\forall\, s, d, r, d', r', m, m'. \\
[c_{(s,d)}\ \mathbf{xmt}\ m] \wedge \mathbf{EF}[c_{(d',r')}\ \mathbf{xmt}\ m'] \wedge m \sim m' \wedge \textit{secret}(s, d, r, m) \rightarrow \\
\textit{legitimate}(s, d', r', m')
\end{array}
$$

Renaming bound variables and moving all quantifiers to begin of formula:

$$
\begin{array}{l}
\forall\, s, d, r, m, s', d', r', m', s'', d'', r'', m''. \\
[c_{(s,d)}\ \mathbf{xmt}\ m] \wedge \textit{secret}(s, d, r, m) \rightarrow \\
\textit{legitimate}(s, d, r, m) \wedge \textit{trusted}((s, d, r)) \rightarrow \\
[c_{(s'',d'')}\ \mathbf{xmt}\ m''] \wedge \mathbf{EF}[c_{(d',r')}\ \mathbf{xmt}\ m'] \wedge m'' \sim m' \wedge \textit{secret}(s'', d'', r', m'') \\
\rightarrow \\
\textit{legitimate}(s'', d', r', m')
\end{array}
$$

This holds trivially if $\neg([c_{(s'',d'')}\ \mathbf{xmt}\ m''] \wedge \mathbf{EF}[c_{(d',r')}\ \mathbf{xmt}\ m'] \wedge m'' \sim m' \wedge \textit{secret}(s'', d'', r', m''))$.

In the other case, we move $[c_{(s'',d'')}\ \mathbf{xmt}\ m''] \wedge \mathbf{EF}[c_{(d',r')}\ \mathbf{xmt}\ m'] \wedge m'' \sim m' \wedge \textit{secret}(s'', d'', r', m'')$ into premise:

$$
\begin{array}{l}
\forall\, s, d, r, m, s', d', r', m', s'', d'', r'', m''. \\
[c_{(s,d)}\ \mathbf{xmt}\ m] \wedge \textit{secret}(s, d, r, m) \rightarrow \\
\textit{legitimate}(s, d, r, m) \wedge \textit{trusted}((s, d, r)) \\
\wedge\ [c_{(s'',d'')}\ \mathbf{xmt}\ m''] \wedge \mathbf{EF}[c_{(d',r')}\ \mathbf{xmt}\ m'] \wedge m'' \sim m' \wedge \textit{secret}(s'', d'', r', m'') \\
\rightarrow \\
\textit{legitimate}(s'', d', r', m')
\end{array}
$$

Since $[c_{(s'',d'')}$ **xmt** $m'']$ ∧ *secret*$(s'', d'', r', m'')$, it also holds that *legitimate*$(s'', d'', r, m'')$ ∧ *trusted*$((s'', d'', r))$:

$\forall\, s, d, r, m, s', d', r', m', s'', d'', r'', m''.$
$[c_{(s,d)}$ **xmt** $m] \wedge$ *secret*$(s, d, r, m) \rightarrow$
*legitimate*$(s, d, r, m) \wedge$ *trusted*$((s, d, r))$
$\wedge\, [c_{(s'',d'')}$ **xmt** $m''] \wedge$ **EF**$[c_{(d',r')}$ **xmt** $m'] \wedge m'' \sim m' \wedge$ *secret*$(s'', d'', r', m'')$
$\wedge$ *legitimate*$(s'', d'', r, m'') \wedge$ *trusted*$((s'', d'', r))$
$\rightarrow$
*legitimate*$(s'', d', r', m')$

Drop no longer required conjuncts:

$\forall\, s, d, r, m, s', d', r', m', s'', d'', r'', m''.$
$[c_{(s'',d'')}$ **xmt** $m''] \wedge$ **EF**$[c_{(d',r')}$ **xmt** $m'] \wedge m'' \sim m'$
$\wedge$ *legitimate*$(s'', d'', r, m'') \wedge$ *trusted*$((s'', d'', r))$
$\rightarrow$
*legitimate*$(s'', d', r', m')$

Insert definition of *trusted* (page 10.1, Definition 10.1) and subsequently definition of *modifies* (page 10.7, Definition 10.7):

$\forall\, s, d, r, m, s', d', r', m', s'', d'', r'', m''.$
$[c_{(s'',d'')}$ **xmt** $m''] \wedge$ **EF**$[c_{(d',r')}$ **xmt** $m'] \wedge m'' \sim m'$
$\wedge$ *legitimate*$(s'', d'', r, m'') \wedge$ *authenticated*$((s'', d'', r))$
$\wedge \neg$*leaks*$((s'', d'', r)) \wedge \neg$*intro*$((s'', d'', r))$
$\wedge \neg$*changes*$((s'', d'', r)) \wedge \neg$*mixes*$((s, d'', r))$
$\wedge \neg$*duplicates*$((s'', d'', r))$
$\rightarrow$
*legitimate*$(s'', d', r', m')$

Apply Lemma 11.1:

$\forall\, s, d, r, m, s', d', r', m', s'', d'', r'', m''.$
$[c_{(s'',d')}$ **xmt** $m''] \wedge$ **EF**$[c_{(d',r')}$ **xmt** $m'] \wedge m'' \sim m'$
$\wedge$ *legitimate*$(s'', d', r, m'') \wedge$ *authenticated*$((s'', d', r))$
$\wedge \neg$*leaks*$((s'', d', r)) \wedge \neg$*intro*$((s'', d', r))$
$\wedge \neg$*changes*$((s'', d', r)) \wedge \neg$*mixes*$((s, d', r))$
$\wedge \neg$*duplicates*$((s'', d', r))$
$\rightarrow$
*legitimate*$(s'', d', r', m')$

Apply Lemma 11.2:

$$
\begin{aligned}
&\forall\, s, d, r, m, s', d', r', m', s'', d'', r'', m'.\\
&[c_{(s'',d')}\ \textbf{xmt}\ m'] \wedge \textbf{EF}[c_{(d',r')}\ \textbf{xmt}\ m'] \wedge m' \sim m'\\
&\wedge\ \textit{legitimate}(s'', d', r, m') \wedge \textit{authenticated}((s'', d', r))\\
&\wedge\ \neg\textit{leaks}((s'', d', r)) \wedge \neg\textit{intro}((s'', d', r))\\
&\wedge\ \neg\textit{changes}((s'', d', r)) \wedge \neg\textit{mixes}((s, d', r))\\
&\wedge\ \neg\textit{duplicates}((s'', d', r))\\
&\rightarrow\\
&\textit{legitimate}(s'', d', r', m')
\end{aligned}
$$

□

We have shown that the confidentiality condition guarantees confidentiality. The confidentiality condition asserts that secret messages are send only if communicating the message is legitimate and if the channel is trusted. In Section 10.2.1 we introduced assumption predicates to model the assumptions of a user/application about a communication path. Next, we show which assumptions have to be correct in order to guarantee the confidentiality condition. We take recourse on the assumption predicates defined in Section 10.2 (Definitions 10.2, 10.9, 10.13, 10.15, 10.11). We introduce the concept of an *attentive party*. An attentive party is aware of (i.e. has the right assumptions) about the state of the communication channel in respect to possible altering of messages, and privacy of the channel. He will send messages only if the channel is private, messages are not altered, and the message is legitimate.

**Definition 11.5** (Attentive Party)**.**

$$
\begin{aligned}
&\textit{AttentiveParty}(s) \equiv\\
&\forall\, d, r, m.[c_{(s,d)}\ \textbf{\textit{xmt}}\ m] \wedge \textit{secret}(s, d, r, m) \rightarrow\\
&\qquad \wedge\ (\textit{asmModifies}(s, (s, d, r)) \leftrightarrow \textit{modifies}((s, d, r)))\\
&\qquad \wedge\ (\textit{asmPrivate}(s, (s, d, r)) \leftrightarrow \textit{private}((s, d, r)))\\
&\qquad \wedge\ \textit{asmIdentitiy}(s, (s, d, r), s, d, r)\\
&\qquad \wedge\ \neg\textit{asmModifies}(s, (s, d, r))\\
&\qquad \wedge\ \textit{asmPrivate}(s, (s, d, r))\\
&\qquad \wedge\ \textit{legitimate}(s, d, r, m)
\end{aligned}
$$

Intuitively, attentiveness means that secret messages are send only if

- it is legitimate to send them,
- the assumptions about the channel in respect to modification of messages, privacy of the channel, and identity of the communicating party are correct,
- and the channel is private and not modifying data.

We show that the confidentiality condition is satisfied if both the user and the application are attentive.

**Theorem 11.2.** *Let $S = system(U, A, I, O, M, u, a, \lambda)$ be a system model.*

$$\mathit{AttentiveParty}(u) \wedge \mathit{AttentiveParty}(a) \rightarrow \mathit{ConfCond}$$

**Proof**

Insert definition of *ConfCond*:

$$\begin{array}{l}\mathit{AttentiveParty}(u) \wedge \mathit{AttentiveParty}(a) \rightarrow \\ \forall\, s, d, r, m. \quad [c_{(s,d)}\ \mathbf{xmt}\ m] \wedge \mathit{secret}(s, d, r, m) \rightarrow \\ \qquad\qquad \mathit{legitimate}(s, d, r, m) \wedge \mathit{trusted}((s, d, r))\end{array}$$

Since $\mathit{secret}(s, d, r, m)$ is true only for $s = u \wedge r = a$ or $s = a \wedge r = u$, we can restrict $s$ and $d$ to $u$ and $a$, respectively. We further restrict $s$ and $r$ to $s = u \wedge r = a$, because the proof for $s = a \wedge r = u$ is identical. Inserting $s = a \wedge r = u$:

$$\begin{array}{l}\mathit{AttentiveParty}(u) \wedge \mathit{AttentiveParty}(a) \rightarrow \\ \forall\, d, m. \quad [c_{(u,d)}\ \mathbf{xmt}\ m] \wedge \mathit{secret}(u, d, a, m) \rightarrow \\ \qquad\qquad \mathit{legitimate}(u, d, a, m) \wedge \mathit{trusted}((u, d, a))\end{array}$$

Insert definition of $\mathit{AttentiveParty}(u)$:

$$\begin{array}{l}\forall\, d, r, m : [c_{(u,d)}\ \mathbf{xmt}\ m] \wedge \mathit{secret}(u, d, a, m) \rightarrow \\ \quad \wedge\, (\mathit{asmModifies}(s, (u, d, r)) \leftrightarrow \mathit{modifies}((u, d, r))) \\ \quad \wedge\, (\mathit{asmPrivate}(s, (u, d, r)) \leftrightarrow \mathit{private}((u, d, r))) \\ \quad \wedge\, \mathit{asmIdentitiy}(u, (u, d, a), u, d, a) \\ \quad \wedge\, \neg \mathit{asmModifies}(s, (u, d, r)) \\ \quad \wedge\, \mathit{asmPrivate}(s, (u, d, r)) \\ \quad \wedge\, \mathit{legitimate}(u, d, r, m) \\ \wedge\, \mathit{AttentiveParty}(a) \rightarrow \\ \forall\, d, m. \quad [c_{(u,d)}\ \mathbf{xmt}\ m] \wedge \mathit{secret}(u, d, a, m) \rightarrow \\ \qquad\qquad \mathit{legitimate}(u, d, a, m) \wedge \mathit{trusted}((u, d, a))\end{array}$$

From $\mathit{asmModifies}(s, (u, d, r)) \leftrightarrow \mathit{modifies}((u, d, r)))$ and $\neg \mathit{asmModifies}(s, (u, d, r))$ it follows that $\neg \mathit{modifies}((u, d, r)))$. From $(\mathit{asmPrivate}(s, (u, d, r)) \leftrightarrow \mathit{private}((u, d, r)))$ and $\mathit{asmPrivate}(s, (u, d, r))$ it follows that $\mathit{private}((u, d, r)))$:

$$\begin{array}{l}\forall\, d, r, m. [c_{(u,d)}\ \mathbf{xmt}\ m] \wedge \mathit{secret}(u, d, a, m) \rightarrow \\ \quad \wedge\, \mathit{asmIdentitiy}(u, (u, d, a), u, d, a) \\ \quad \wedge\, \neg \mathit{modifies}((u, d, r)) \\ \quad \wedge\, \mathit{private}((u, d, r)) \\ \quad \wedge\, \mathit{legitimate}(u, d, r, m) \\ \wedge\, \mathit{AttentiveParty}(a) \rightarrow \\ \forall\, d, m. \quad [c_{(u,d)}\ \mathbf{xmt}\ m] \wedge \mathit{secret}(u, d, a, m) \rightarrow \\ \qquad\qquad \mathit{legitimate}(u, d, a, m) \wedge \mathit{trusted}((u, d, a))\end{array}$$

Moving quantifiers, renaming bound variables, and moving conjunction $[c_{(u,d)}$ **xmt** $m] \wedge$ *secret*$(u, d, a, m)$ into premise:

$$
\begin{array}{l}
\forall\, d, r, m, d', m'. \\
[c_{(u,d')} \textbf{ xmt } m'] \wedge \textit{secret}(u, d', a, m') \\
\wedge\, ([c_{(u,d)} \textbf{ xmt } m] \wedge \textit{secret}(u, d, a, m) \rightarrow \\
\qquad \wedge\, \textit{asmIdentitiy}(u, (u, d, a), u, d, a) \\
\qquad \wedge\, \neg \textit{modifies}((u, d, r)) \\
\qquad \wedge\, \textit{private}((u, d, r)) \\
\qquad \wedge\, \textit{legitimate}(u, d, r, m)) \\
\wedge\, \textit{AttentiveParty}(a) \rightarrow \\
\textit{legitimate}(u, d', a, m') \wedge \textit{trusted}((u, d', a))
\end{array}
$$

It remains to be shown that

$$
\begin{array}{l}
\textit{asmIdentitiy}(u, (u, d, a), u, d, a) \\
\wedge\, \neg \textit{modifies}((u, d, a)) \\
\wedge\, \textit{private}((u, d, a)) \\
\wedge\, \textit{AttentiveParty}(a) \\
\rightarrow \\
\textit{trusted}((u, d, a))
\end{array}
$$

Inserting definition of *trusted* (Definition 10.1in Section 10.2.1):

$$
\begin{array}{l}
\textit{asmIdentitiy}(u, (u, d, a), u, d, a) \\
\wedge\, \neg \textit{modifies}((u, d, a)) \\
\wedge\, \textit{private}((u, d, a)) \\
\wedge\, \textit{AttentiveParty}(a) \\
\rightarrow \\
\textit{authenticated}((u, d, a)) \wedge \neg \textit{leaks}((u, d, a)) \\
\wedge\, \neg \textit{modifies}((u, d, a))
\end{array}
$$

We show separately that

$$
\begin{array}{l}
\textit{asmIdentitiy}(u, (u, d, a), u, d, a) \wedge \textit{AttentiveParty}(a) \rightarrow \\
\textit{authenticated}((u, d, a))
\end{array}
$$

and

$$
\textit{private}((u, d, a)) \rightarrow \neg \textit{leaks}((u, d, a))
$$

First proof:

$$\textit{asmIdentitiy}(u,(u,d,a),u,d,a) \wedge \textit{AttentiveParty}(a) \rightarrow \\ \textit{authenticated}((u,d,a))$$

From the definition of *AttentiveParty*$(a)$ follows *asmIdentitiy*$(a,(u,d,a),u,d,a)$. Inserting definition of *authenticated*$((u,d,a))$:

$$\textit{asmIdentitiy}(u,(u,d,a),u,d,a) \wedge \textit{asmIdentitiy}(a,(u,d,a),u,d,a) \rightarrow \\ \textit{asmIdentitiy}(u,(u,d,a),u,d,a) \wedge \textit{asmIdentitiy}(a,(u,d,a),u,d,a)$$

□

Second proof:

$$\textit{private}((u,d,a)) \rightarrow \neg \textit{leaks}((u,d,a))$$

Inserting definitions of *private* and *leaks*:

$$[c_{(s,d)}\ \textbf{xmt}\ m] \wedge \textbf{EF}[c_{(d',r')}\ \textbf{xmt}\ m'] \wedge m \sim m' \rightarrow \\ d = d' \wedge r = r' \\ \rightarrow \\ \neg(\ \exists m, m', d', r'. \\ \quad [c_{(s,d)}\ \textbf{xmt}\ m] \wedge \textbf{EF}[c_{(d',r')}\ \textbf{xmt}\ m'] \\ \quad \wedge m \sim m' \wedge (d \neq d' \vee r \neq r'))$$

Applying De Morgan's law:

$$[c_{(s,d)}\ \textbf{xmt}\ m] \wedge \textbf{EF}[c_{(d',r')}\ \textbf{xmt}\ m'] \wedge m \sim m' \rightarrow \\ d = d' \wedge r = r' \\ \rightarrow \\ \forall m, m', d', r'.\ \neg[c_{(s,d)}\ \textbf{xmt}\ m] \vee \neg\textbf{EF}[c_{(d',r')}\ \textbf{xmt}\ m'] \\ \qquad\qquad \vee \neg m \sim m' \vee (d = d' \wedge r = r')$$

□

# Chapter 12

# Summary

In this part of the thesis, the formal methodology developed in Part I has been used to develop formal definitions of security in Human-Computer Interaction. A generic system model has been introduced in Chapter 9. The generic system model defines the basic components of our formal modeling approach and the ways in which these basic components can possibly interact. It is flexible enough to describe a large class of human-computer interactions on the one hand, while on the other hand it contains enough structure to model details of interaction. The formal methodology introduced in Part I has been extended by *colored messages*. With colored messages, it becomes possible to specify component behavior in terms of message traces. In Chapter 9, a generic formal model of interactive systems has been introduced. The model introduced allows to model interaction between multiple users and applications, using multiple input and output devices. The *core predicates* introduced in Section 9.2 provide a set of predicates for the specification of security properties in the generic system model. Based on the generic system model and the core predicates, the concepts of the Common Criteria for Information Technology Security Evaluation (CC) relevant for human-computer interaction have been formalized. The formalization of Common Criteria concepts was achieved in two steps: For each category of Common Criteria concepts, first a number of core concepts were identified. The formal definitions of these core concepts served as building locks for the formal definition of the Common Criteria concepts.

The Common Criteria are a collection of security requirement definitions. A software engineer selects the relevant concepts for the application at hand. The Common Criteria are a valuable collection of security concepts and provide a (semi-formal) methodology for the certification of security products against an international standard. It does not contain the provisions for guaranteeing pervasive security in human-computer interaction, because a) it does not guarantee that all aspects of security are covered, and b) it does not require the pervasive use

of formal methods even at the highest evaluation level EAL 7. It require a formal presentation of the functional specification and high-level design and a formal and semiformal demonstration of the correspondence between the high-level and the low-level design, but correspondence with the actual implementation is shown by extensive testing only.

These shortcomings are overcome in Chapter 11. A computer system is considered secure if it guarantees *Confidentiality*, *Integrity*, and *Availability*. These generic concepts are adapted to human-computer interaction and formalized. In a second step, we show how the core definitions developed for the formalization of Common Criteria concepts are also suitable as building locks for breaking down CIA concepts into sub-concepts. We have shown that the same building locks can be used to describe formal computer security concepts both in terms of the Common Criteria and in terms of Confidentiality, Integrity, and Availability. The break-down of the Confidentiality concepts developed in Section 11.2 will be used in Chapter 15 in the specification and verification of confidentiality of a secure email client. In Chapters 14 and 16, we show for the email client that it satisfies integrity and availability as defined in Section 11.1, too.

# Part III

# Specification and Verification of Secure Applications

—

# The Verisoft Email Client

# Chapter 13

# Specifying and Verifying a Secure Email System

## 13.1 Introduction

In this part, the methodology developed in Parts I and II is applied to the specification of a secure email client. We show how compliance of the specification of an email client to the security requirements defined in Part II can be guaranteed. In the Verisoft project, an actual implementation of the secure email client has been written and verified. In this part, we do not only show that the methodology from the previous parts is applicable to real-world software systems. We also develop a set of design patterns which can be used in other applications. In Chapter 14, we develop a design pattern for text-based, interactive application and show that application specifications following the pattern satisfy the security requirement of system integrity. This design pattern is applicable to all text-based, interactive applications. The confidentiality condition developed in Chapter 15 is applicable to all user and application models; independent of a concrete system, confidentiality is guaranteed if the confidentiality condition is satisfied by the communicating parties.

The methodology developed in this thesis has been applied to the development of a secure email client as part of the Verisoft project. Verisoft is a long term research project funded by the German Federal Ministry of Education and Research (BMBF). With 12 partners from German industry and academia and a funding of 20 million Euro for seven years, Verisoft and its successor Verisoft XT is one of the biggest software verification projects in Germany.

The goal of the Verisoft project was to create the tools and methods to allow the pervasive formal verification of computer systems, and to show that verification of real world systems is viable (Paul, 2005). In Verisoft, formal methods and

verification technology have been used throughout all aspects of system developing, including verified hardware, verified development tools, and verified operating systems and verified application programs. Four concrete systems were developed in Verisoft. Of these four systems, three were developed by or in cooperation with partners from industry, and one is developed by the academic partners. The industry projects include an *Emergency Call System* developed in cooperation with the BMW group, a *Biometric Identification System* in cooperation with T-Systems, and *Hardware verification* developed in cooperation with Infineon Technologies. The academic project develops a secure email system. In this part of the thesis, we show how our methodology for the development of secure interactive systems has been used in the specification and verification of that system.

### 13.1.1 The Academic System

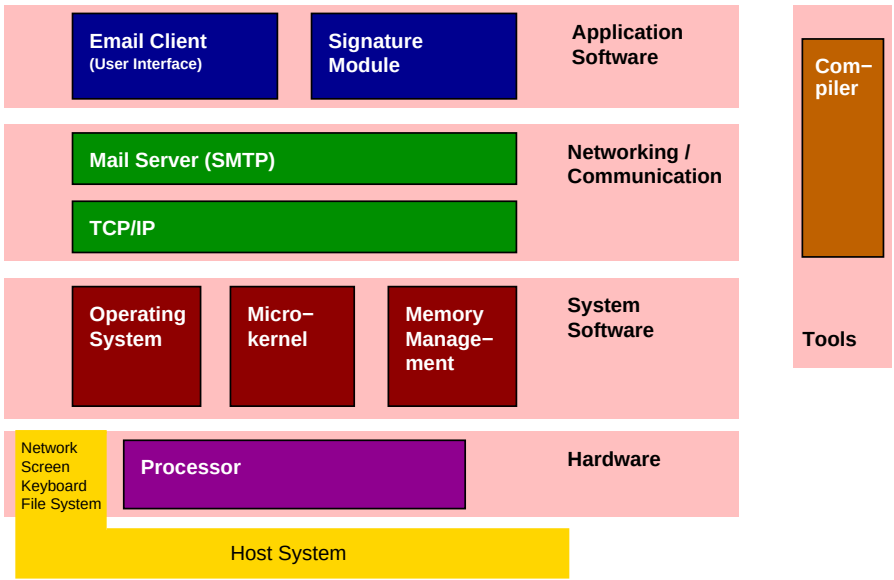


Figure 13.1: Components of the academic system (Beuster et al., 2006)

The goal of the academic subproject was to show that common desktop technology can be formally specified and verified. For this reason, the technology used in the academic system stays as close to off-the-shelf desktop systems, technologies, and standards as possible. The academic system is made up of different parts, as depicted in Figure 13.1. The verified compiler accepts programs written in the C dialect C0 by Leinenbach et al. (2005). The machine code is run on fully verified hardware (processor) by Ayewah et al. (2005). Three layers of software build upon the hardware. The first layer consists of a fully verified micro-kernel,

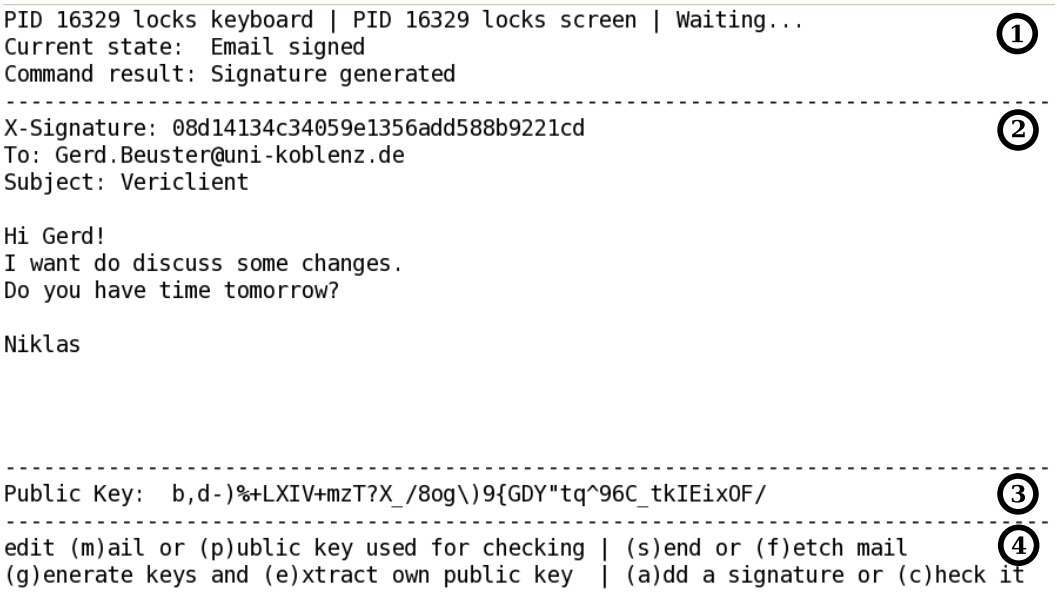


Figure 13.2: Vericlient prototype running: The numbers indicate the following screen areas: (1) Status / current state of the email client (2) Editing area (3) Public key (4) Commands available (Beuster et al., 2006)

memory management unit and an accompanying operating system called *Simple Operating System*, developed by Gargano et al. (2005). The networking and communication layer consists of a fully verified SMTP mail server using a fully verified TCP/IP stack. This allows the academic Verisoft system to interconnect with the real world like intranets or the Internet. The application software sits on top of the system software and the communication layer.

As part of the Academic Verisoft System, the subproject carried out at University Koblenz-Landau developed a completely verified email client. The formal specification of the email client includes all informal requirements and security goals. Compliance to the formal specification has been proven for the complete source code. The Verisoft Email Client consists of approximately 100 procedures, totaling 4000 lines of code. The formal specification and verification in Isabelle/HOL (Schirmer, 2005) consists of approximately 16.000 lines. The Verisoft email client has been developed and verified in three man years. Specification, source code, and proof scripts are freely downloadable from the Verisoft Repository at `http://www.verisoft.de/VerisoftRepository.html`.

Within the academic part of the Verisoft project, the Verisoft email client, for short *Vericlient*, provides the interface to the user. When a user accesses the aca-

demic system, he interacts with the email client. The email client itself has internal interfaces to four components: The I/O facilities (via the operating system), the SMTP server for delivery and reception of emails, and the signature component for generation and checking of signatures. For its internal operation, the email client makes use of data structures of a C library developed by Starostin (2006).

Providing a user interface is the core functionality of the email client. The design goal of the Verisoft email client was to provide the core functionality of an email client, plus the possibility to handle digital signatures. The Verisoft email client provides a full screen text editor for reading and writing email, and for editing the public keys used for checking signatures. An interfaces to the SMTP component of the Verisoft email system allows to send and receive mail. An interface to the Verisoft signature component allows to generate public/private key pairs, sign messages with the generated private key, insert the generated public key into the message (in order to send it to a recipient), and to check email messages against public keys entered by the user. All functionality is accessible via a TTY interface. The email client has two modes: In command mode, certain characters entered via the keyboard are interpreted as commands. For example, key 'a' signs a message, and key 'f' fetches the next unread email from the server. In edit mode, the email message and the public key are edited in a full screen editor. The email client does not provide means for managing email folders. There is only one email in the system at any point in time. The user interface provided by the Verisoft email client is shown in Figure 13.2. Details of the interface are explained in the next chapters.

Both functional correctness and security have been proven for the secure email client. In the following three chapters, we show how correctness of the Verisoft email client specification in respect to the three core principles of Confidentiality, Integrity, and Availability, as defined in Chapter 11, has been established. Section 14 shows how Integrity (Definition 11.2 in Chapter 11) is guaranteed. Chapter 15 shows that the specification of the Verisoft email client satisfies Confidentiality as defined in Definition 11.1 in Chapter 11. Finally, Chapter 16 ensures Availability (Definition 11.3 in Chapter 11) of the email client.

## 13.2 Related Projects

Another important fundamental research project in the area of verification and analysis is the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (AVACS), which is funded by the Deutsche Forschungsgemeinschaft (DFG). About 70 scientists of the Universities of Oldenburg, Freiburg and Saarbrücken as well as the Max-Planck-Institute for Computer Sciences in Saarbrücken are working on the improvement of techniques

for mathematically precise verification, including the development of tools. The goal of their work is to automate safety analyses of critical embedded systems which are used for example in aircrafts, motor vehicles or railway transportation (Damm et al., 2004).

Significant prior projects are DAEDALUS and VERIFIX. The DEADALUS consortium was a research and technology development project in the Fifth Framework Programme (FP5) of the European Union. With the contributions of universities from France, Germany, Denmark, and Israel, the project developed methods and tools to support the industrial validation of critical concurrent software by static analysis and abstract testing (Goubault, 2001; Cousot and Cousot, 2002). The goal of VERIFIX, another project funded by DFG, was the construction of mathematically correct compilers, which included the development of formal methods for specification and implementation of a compiler. One of the project's results was a fully verified LISP interpreter (Goos and Zimmermann, 1999). The scope of Verisoft goes beyond DAEDALUS and VERIFIX. In difference to DAEDALUS, the systems developed in Verisoft are pervasively verified. In difference to VERIFIX, not only a compiler, but complete systems are verified.

In project Bang 3, we have used formal method for the specification of components of multi-agent systems (MAS), and for reasoning about properties of MAS (Beuster et al., 2003, 2004; Beuster and Neruda, 2006).

# Chapter 14

# Secure Interaction and Information Display

Wrong assumptions about the state of the computer system are a main source of error in human-computer interaction (HCI). In this chapter, we show how Integrity (as defined in Definition 11.2 in Chapter 11) is guaranteed. We show how consistency requirements between the state of a computer system and the user's assumptions about the state can be defined formally. We show that the main execution loop introduced in Algorithm 3 in Chapter 7 violates integrity constraints. A improved main execution loop is introduced. We show that the improved execution loop satisfies the integrity constraints. Furthermore, we give first definitions of the main execution loop functions `updateScreen` and `execute`. These definitions, which are refined in the next chapters, close the gap between the high-level application specification with IOLTS and CTL on the one hand, and the low level specification of program procedures on the other hand, as described in Chapter 7.

## 14.1 Introduction

### 14.1.1 The Problem

Informally, a system is consistent if the user's assumptions about the system correspond to the actual system state whenever he interacts with the system. There are two main sources for wrong assumptions about the system state:

***Inconsistency during updates.*** Human-Computer Interaction (HCI) is inherently asynchronous. Execution of user commands and updates of the data displayed by the output device take time. Due to the inherently asynchronous character of Human-Computer Interaction, the user may err about the system state; either because commands have not been executed yet, or because

the screen has not been updated.

***Insufficient data or wrong interpretation of data.*** The system may not provide enough information to determine the system state, or the user may interpret application output wrongly. A large part of the specification of interactive applications is concerned with the relation between user input and the information shown to the user. For example, when editing a text, the current (internal) state of the text should be shown to the user, and user input should cause corresponding changes to the text. Usually, the specification of user input and system output is rather informal. Specifications declare that something (e.g., a text) "is shown on the screen" and the user "enters a text." In most cases, this informal description is sufficient. However, in critical applications, a precise and formal definition is desirable.

The latter source of inconsistencies, wrong interpretation of data, has been addressed frequently. For example, Reeder and Maxion (2005) analyzed the problem of representing NTFS file permissions on Windows XP systems and developed the design principle of "anchor-based subgoaling" in order to mitigate the problem.

Here, we concentrate on the former of the mentioned sources of errors, namely inconsistencies during updates. Most user interface security requirements are highly application-specific. However, there are also some generic requirements. We show that for a large class of applications, it is possible to define generic requirement in a formal way. In this chapter, we focus on one of these generic requirements: The user should always be aware of the system state when issuing a command. We show how consistency during updates can be guaranteed for text-based applications.

### 14.1.2 Plan of This Chapter

In Section 14.2, we show that the common approach to modeling interactive applications does not guarantee consistency. We provide an alternative model for which consistency can be guaranteed.

Integrity is defined in CTL, and model checking is used to show that a component given as an IOLTS satisfies the Integrity constraint. In order to pervasively specify and verify a critical application, it is also necessary to describe program behavior with pre- and post-conditions for concrete procedures that are part of the system. We use Hoare logic for this. In two steps, we close the gap between the more abstract state-based modeling on the one hand and more concrete pre-/post-condition-based modeling on the other hand. Based on the IOLTS of the main execution loop developed in Section 14.2 we provide a generic code template for the concrete implementation of the main execution loop of abstract state-based

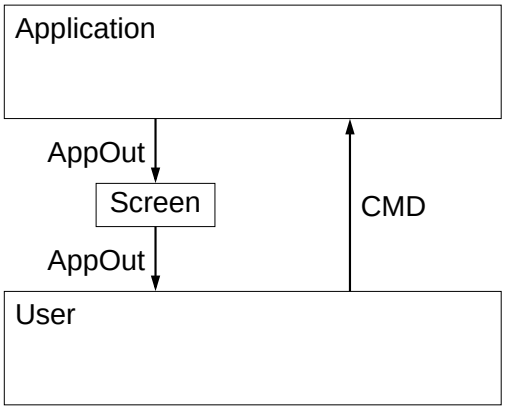


Figure 14.1: Basic system model (user + application).

models in Section 14.3. A method for the integration of state-based formal modeling methods and pre-/postcondition based methods has been proposed in Chapter 7. In Sections 14.3.2 and 14.3.3 this integration method is demonstrated in the specification of `execute` and `updateScreen` for the editor sub-component.

## 14.2 Guaranteeing Integrity

In Chapter 9, a generic system model has been developed. Other applications and users are relevant for guaranteeing confidentiality between the user and the application, but they are not relevant for integrity. Therefore, we start by using a simplified version of the system model from Figure 9.2 (Section 9.2). In the simplified version, shown in Figure 14.1, the user interacts directly with the application. Since the keyboard component just relays input from the user to the application component, it is not relevant for the integrity properties. The screen device, however, is relevant, because it is an "asynchronous" component.

Two types of messages are used to exchange information between the user and the application: *AppOut* is the data type for information shown on the screen. *CMD* is the data type for input given by the user. The generic system model can be further structured without losing generality. All well-designed applications (and all reasonable models of user behavior) split up the components into a generic execution loop, governing the general behavior of the application (or the user), and an application (task) specific component. The separation of a generic execution loop and a task specific component serves two purposes: Firstly, it follows established system design practice and therefore allows realistic modeling of applications. Secondly, the separation into a generic and an application-specific

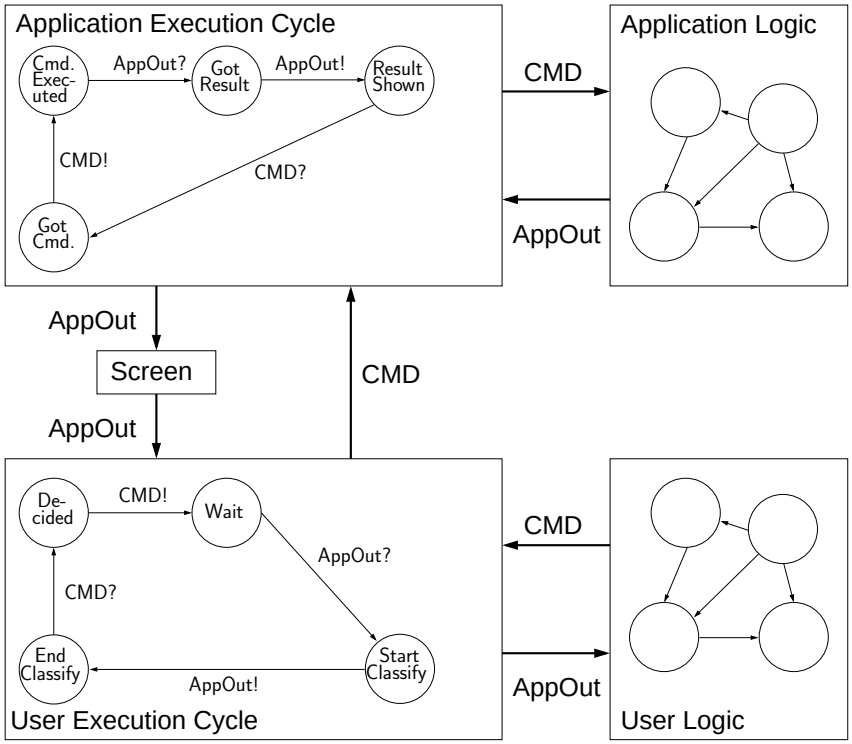


Figure 14.2: Basic model of user and application.

component allows to deduce properties that hold for all applications with this design, independently of the concrete application's task and application logic.

A basic model following this approach is shown in Figure 14.2. In this model, *AppOut* and *CMD* are variables representing all possible command input and application output. Question marks after variable names indicate reading of an input value, and exclamation marks indicate writing of an output value. Thus, in one cycle of application execution, the following steps are taken (note that the same message, e.g. a command, can be passed between different components):

1. The application accepts a command from the user:

$$\text{ResultShown} \xrightarrow{\text{CMD?}} \text{GotCommand}$$

2. The command is passed to the application logic for processing:

$$\text{GotCommand} \xrightarrow{\text{CMD!}} \text{CommandExecuted} \xrightarrow{\text{AppOut?}} \text{GotResult}$$

3. The result of the computation is forwarded to the output device:

$$\text{GotResult} \xrightarrow{\text{AppOut!}} \text{ResultShown}$$

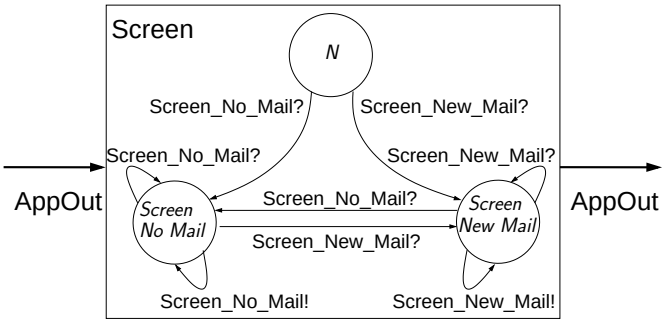


Figure 14.3: Screen component for the email system example.

In a similar way, the user reads an application output, evaluates which command should be issued next, and enters the command into the input device.

The Screen component is an explicit part of the model (see Fig. 14.2). It takes input from the application and presents it to the user. Since the screen is constantly refreshed, there is no simple one-to-one relation between messages sent by the application and messages received by the user. The screen outputs its current content until the content changes. A formal definition of the Screen component is given below, and a graphical representation is given in Figure 14.3.

**Definition 14.1** (Screen Component)**.** *Let $L_a = (S_a, \Sigma_a, s_{0\,a}, \rightarrow_a)$ be an IOLTS specifying an application. A IOLTS $L_s = (S_s, \Sigma_s, s_{0\,s}, \rightarrow_s)$ is a* Screen *component if the following holds, where $n$ is a new symbol with $n \notin \Sigma!$:*

- $S_s = \Sigma!_a \cup \{n\}$
- $s_{0\,s} = n$
- $\Sigma?_s = \Sigma!_a$
- $\Sigma!_s = \Sigma!_a$
- *For all $s, s' \in S_s$:* $s \xrightarrow{s'?} s'$
- *For all $s \in S_s \backslash \{n\}$:* $s \xrightarrow{s!} s$

The screen component has one state for each possible application output message; in this case, `ScreenNoMail` and `ScreenNewMail`. Whenever the application updates the screen, the screen components enters the state associated with the current message. The component repetitively outputs this message ("refreshs the screen") until a new message arrives.

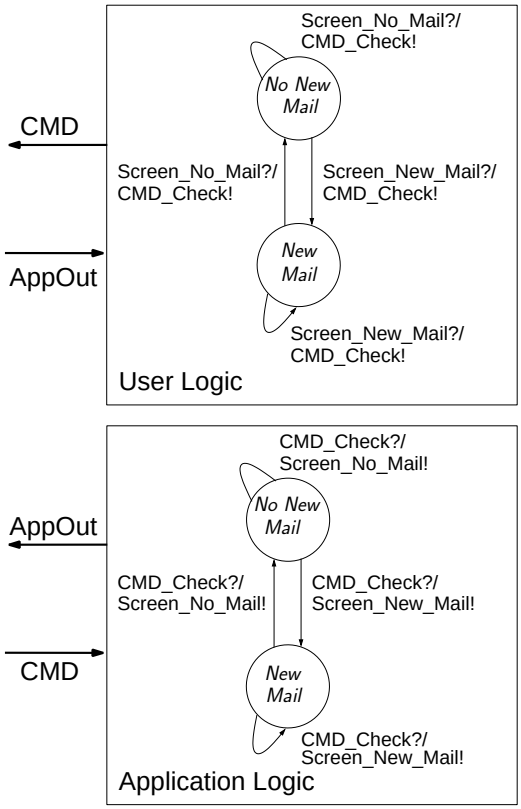


Figure 14.4: Simple user logic and application logic components.

Our basic model already allows to deduce interesting properties with respect to integrity constraints. This is illustrated by the following simple example.

For our example, we make the (reasonable) assumption that the critical states are exactly those where the user makes a decision, i.e.:

$$\lambda(\mathit{critical}) = \{\text{End\_Classify}\}$$

Critical properties of an application and user assumptions about such properties, of course depend on the user logic and the application logic components. The specification of the application logic component is introduced in Chapter 16. Here, we use an excerpt of the full model given in Figure 14.2. The excerpt allows to deduce the relevant properties for guaranteeing integrity while the model checker input and output are still human-readable.

The application starts in a configuration represented by the state “No New Mail.” When the command “Check Mail” (CMD_Check) is received by the application, it may either transit into the state “New Mail” or stay in the state “No New

Mail." In the same way, the command "Check Mail" in state "New Mail" may either lead into the state "No New Mail" or into the state "New Mail." Whether new mail arrived or not is outside of the scope of the component. Therefore, the application logic component switches nondeterministically between states `NewMail` and `NoNewMail`. We assume the user uses the same logic (i.e., we assume that the user "knows" how the application works). The two logic components are shown in Figure 14.4. Since the state names are identical in both components, we use the notion `ApplicationLogic.`$\langle$*state*$\rangle$ when referring to states of the application logic component, and `UserLogic.`$\langle$*state*$\rangle$ when referring to states of the user logic component. As a security relevant property, we define that the user should always know whether new mail has arrived or not:

$$\begin{aligned} \lambda(a_0) &= \{\text{ApplicationLogic.NewMail}\} \\ \lambda(u_0) &= \{\text{UserLogic.NewMail}\} \end{aligned}$$

Representations of the components suitable for model checking with model checker NuSMV (Cimatti et al., 2002) are given in Appendix C.1. If the user logic component and the application logic component are directly connected, both modules are always in corresponding states. This is shown by model checking the system given in Appendix C.4.1.

Integrity is *not* guaranteed if the connection between the user logic component and the application logic component is mediated by the user execution component and the application execution component given in Figure 14.4 (NuSMV code for this component is provided in Appendix C.1.1 and C.1.2).

The problem lies in the lack of consistency, as the trace given in Figure 14.5 shows: When the user decides about the next command for the second time, he does not recognize that the screen output does not reflect the current configuration of the application, but the previous one. In step 1, the application outputs that no new mail has arrived. This is shown on the screen in step 2, where the user sees it. In steps 3 to 4, the user decides to check for new mail. The command is issued by the user and received by the application in step 5. The application checks for new mail again in step 6. New mail has arrived, therefore the application logic is in state "New mail" in step 7. In step 8, the user is not aware that the screen has not been updated yet. Therefore, he still assumes that no new mail has arrived in step 9 and also in step 10.

It should be noted that the IOLTSs we use for modelling are still synchronous automata. The asynchronicity effect leading to inconsistency comes from the way the screen output component is modeled (it is explicitly modeled using a synchronous formalism).

| Step | Application Execution | User Execution | Application Logic | User Logic |
|---|---|---|---|---|
| 1 | Got Result | Wait | No new mail | No new mail |
| App. Exec: | Got Result | —Screen_No_Mail!→ | Result Shown | |
| Screen: | N | —Screen_No_Mail?→ | Screen No Mail | |
| 2 | Result Shown | Wait | No new mail | No new mail |
| Screen: | Screen No Mail | —Screen_No_Mail!→ | Screen No Mail | |
| User Exec.: | Wait | —Screen_No_Mail?→ | Start Classify | |
| 3 | Result Shown | Start Classify | No new mail | No new mail |
| User Exec.: | Start Classify | —Screen_No_Mail!→ | End Classify | |
| User Logic: | No new mail | —Screen_No_Mail?/CMD_Check!→ | No new mail | |
| 4 | Result Shown | End Classify | No new mail | No new mail |
| User Exec.: | End Classify | —CMD_Check?→ | Decided | |
| 5 | Result Shown | Decided | No new mail | No new mail |
| User Exec.: | Decided | —CMD_Check!→ | Wait | |
| App. Exec.: | Result Shown | —CMD_Check?→ | Got Command | |
| 6 | Got Command | Wait | No new mail | No new mail |
| App. Exec.: | Got Command | —CMD_Check!→ | Cmd. Executed | |
| App. Logic: | No new mail | —CMD_Check?/Screen_New_Mail!→ | New mail | |
| 7 | Cmd. Executed | Wait | New mail | No new mail |
| App. Exec: | Cmd. Executed | —Screen_New_Mail?→ | Got Result | |
| 8 | Got Result | Wait | New mail | No new mail |
| Screen: | Screen No Mail | —Screen_No_Mail!→ | Screen No Mail | |
| User Exec.: | Wait | —Screen_No_Mail?→ | Begin Classify | |
| 9 | Got Result | Begin Classify | New mail | No new mail |
| User Exec.: | Begin Classify | —Screen_No_Mail!→ | End Classify | |
| User Logic: | No new mail | —Screen_No_Mail?/CMD_Check→ | No new mail | |
| 10 | Got Result | End Classify | New mail | No new mail |

Figure 14.5: Excerpt of a trace refuting consistency of the basic model (see Fig. 14.2).

In Definition 11.2 (Chapter 11), Integrity has been defined as

$$\textit{Integrity} \equiv$$
$$\textit{appCritical} \rightarrow ((a_0 \leftrightarrow u_0) \wedge (a_1 \leftrightarrow u_1) \wedge \ldots \wedge (a_n \leftrightarrow u_n))$$

with $a_0, \ldots, a_n$ the attributes representing the configuration of the application, $u_0, \ldots, u_n$ the user's representation of these attributes, and *appCritical* whenever the application is in a critical state.

The system model does not satisfy the integrity constraint, because the user may decide about which command to issue based on the screen output of an older state of the application logic. The user may then choose a wrong command. In particular, if he does not know whether a previous command has already been executed, he may be tempted to re-issue the same command. In the worst case, this can lead to a security problem, for example when the user accidentally confirms a critical action twice.

Next, we show how the synchronization problem illustrated by the above example can be solved.

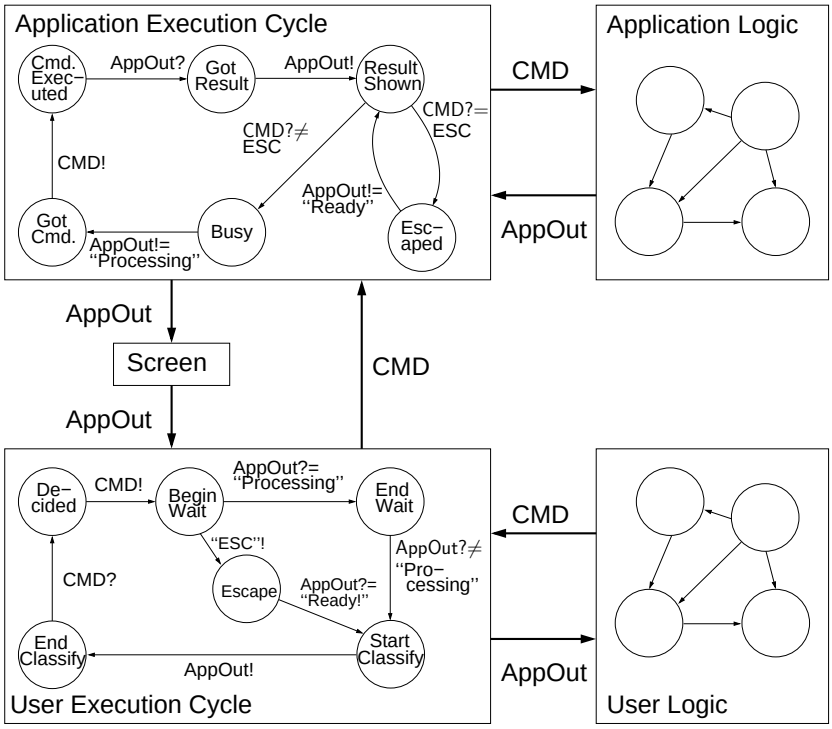


Figure 14.6: Improved model with states for synchronization.

The problem can be solved by introducing new states for synchronization, as shown in Figure 14.6. In this model, the application gives visual feedback indicating whether it is waiting for user input or processing user input. Once the

application has received a user command, it shows “processing” on the screen. When processing is finished, the new application status is shown. However, just showing the message “processing” while executing user commands is not sufficient. Depending on execution speed, the user may not recognize the message “processing” at all (because it was shown only for a very short time), or it may take a long time before the message is shown (in case the system is slow). In order to give the user the ability to distinguish between the two cases, an escape command is introduced. If the user gives the escape command, the message “Ready!” is shown. This way, if the user does not know about the state of the input process, he can press “escape” and wait for the message “Ready!” to show up.

One can verify using model checking, that the improved model given in Figure 14.6 satisfies the integrity constraint. Definitions of the improved application execution component, user execution component, and system component suitable for model checking with NuSMV are given in Appendix D.

We have shown that the naïve model of user and application interaction is not sufficient to guarantee consistency. While we presented an improved model guaranteeing consistency for the given application logic and user logic components, we did not—and can not—show that the consistency constraint holds for *all possible* application and user logic components. Consistency does not solely rely on the application and user execution loops. The user must also make the right assumptions about the application model. He must have knowledge about (an abstraction of) the inner workings of the application, and about the consequences of his actions. This knowledge is represented in the user logic component. Just like the user’s and the application’s execution loop, the logic components can be modeled as IOLTSs. This requires a state-based representation of the application, and of the user’s knowledge about the application.

While it is perfectly fine to improve the specification of the application, one may ask whether it is acceptable to change the user model, i.e., our assumptions about the user, as the user cannot be “re-implemented.” To a certain degree, however, that is possible. It is common practice to train the user on how to operate a system. For this, a formal user model allows to explicit state what a user has to know in order to operate the system.

## 14.3 Improved Main Execution Loop

In Section 14.2, simple application logic and user logic components were used in the refutation of the naïve basic model. These example components (given in Figure 14.4) each have only two configurations: “New Mail” and “No New Mail.” In actual TTY-based applications, we have screens with multiple rows and columns, where each cell can contain an alphanumeric character. Even on a

moderately sized screen, the set of all possible combinations of output characters is too large to be modeled explicitly. Therefore, it is necessary to find a suitable abstraction of application states, application output, and user assumptions about application states in order to make models of real-world applications suitable for automated model checking.

### 14.3.1 Notation

**Notation** We use the Object Constraint Language (OCL) for specification. The OCL constraints given here should be understandable without deeper knowledge of OCL.[1] In a post-condition, `x` refers to the value of attribute or variable `x` after execution of the procedure, and `x@pre` refers to the value of `x` when the procedure was entered. In this, we follow the common OCL syntax. OCL has the shortcoming that it does not make any assumptions about system properties that are not explicitly modeled (frame problem). To solve this problem, we (implicitly) add the following to our specifications: All functions cause only those effects explicitly mentioned. See (Warmer and Kleppe, 1999, 1998) for more information on OCL and (Object Modeling Group) for the current language specification.

As usual in OCL, we refer to the result of a function call by "result" in post-conditions. When functions refer to one- or two-dimensional lists or strings, we use the usual []-notation to refer to elements. That is, $string$[0] is the first character of $string$, $string$[1] is the second, and so on.

### 14.3.2 Main Execution Loop

We assume that in the concrete program, the current state is represented by the variable `state`. User commands (usually corresponding to keystrokes) trigger command execution. Depending on the current state, the command, and the result of a command, the system changes to a new state. The actual specification of the command `execute` is application-dependent. The relationship between the high-level IOLTS model and the specifications of `nextState` and `execute` is defined in Chapter 7. Pseudo code for a main event loop implementing the Application Execution Cycle model from Figure 14.6 is given in Algorithm 4.

Under the assumption that the user knows the program state if it is always given explicitly on the screen, the screen update function `updateScreen` can be specified with the following auxiliary functions:

[1] To make the constraints easier to understand for readers not familiar with OCL, we sometimes use the standard mathematical notation instead of the OCL notation. For example, we use $x \in list$ instead of $list \rightarrow contains(x)$.

**Algorithm 4** The main event loop

```
 1: repeat
 2:    {Show Result}
 3:    updateScreen(status, waiting, cmdResult)
 4:    {Get Command}
 5:    repeat
 6:       cmd := getKeystroke()
 7:       if cmd = ESC then
 8:          {Escaped}
 9:          updateScreen(status, waiting, cmdResult)
10:       end if
11:    until cmd ≠ ESC
12:    {Busy}
13:    updateScreen(status, processing, cdmResult)
14:    {Execute & Get Result}
15:    cmdResult := execute(state,cmd)
16:    state := nextState(state,cmd,cmdResult)
17: until cmd = QUIT
```

- *stateAsString*(`state`) is a string that allows the user to identify the state of the application.

- *screenOutput*(`cmdResult`) is a two-dimensional array of characters. It contains the correct screen output corresponding to the current configuration of the application. The actual definition of `screenOutput` is under the discretion of the application at hand.

- *stringAt*$(x, y)$ is the string shown on screen position $(x, y)$.

We require that the current state of the application logic component plus optionally the additional information "ready" or "processing" are shown in the first line of the screen. A specification of function `updateScreen` in OCL is given in Table 14.1. The specification of `updateScreen` is a generic template. It fits every applications requiring a secure, text-based user interface. The application specific part is of the specification is provided by mathematical function *screenOutput*. An excerpt[2] of the definition of *screenOutput* for the Verisoft email client is given in the next Section.

[2]We only show the part of *screenOutput* relevant to showing the email message. The reader is referred to (Beckert et al., 2007) for a complete specification of *screenOutput.*

| | |
|---|---|
| **context** | `updateScreen(status, flag, cmdResult)` |
| **post** | *stringAt*$(0,0)$ = *stateAsString*(`status`) and<br>**if** `flag = waiting` **then**<br>*stringAt*$(0,1)$ = "Ready"<br>**else**<br>*stringAt*$(0,1)$ = "Processing"<br>**end if**<br>and<br>$\forall k \in \{2,\ldots,$*screenHeight*$-2\}$ :<br>*stringAt*$(0,k)$ =<br>*screenOutput*(`cmdResult`)$[k-2]$ |

Table 14.1: Specification of the application's function for updating the screen contents

### 14.3.3 Editor Component

Not only the state of the system, but also the data has to be displayed correctly. Defining "correct" display of an email under security aspects is a challenging task. In the real world, "phishing" attacks are a major form of electronic fraud (Bachfeld, 2005). Many of these attacks are based on exploitation of incorrect or ambiguous display of email messages. For the Verisoft email client, these attacks are prevented by restricting the way emails are displayed. The Verisoft email client shows the pure ASCII representation of the email.

In the following, we present an excerpt of the specification of the Verisoft email client. This allows us to demonstrate how an interactive user interface component can be specified. The email viewing and editing component has the following characteristics: It is a full screen editor; the user can roam freely over the text using the cursor keys. The text edited may not fit the screen. In that case, the editor will scroll when the cursor reaches the screen borders.

The email message editing field is represented by a data structure *m := (st, cx, cy, co, ro)* with $s$ a list of strings where each element represents a line of the text, $(cx, cy)$ the cursor position and $(ro, co)$ row and column offsets. If the text is larger than the size of the screen, the offsets indicate which part of the email are shown.

The part of the main execution loop's `updateScreen` responsible for showing the email (with $(x, y)$ a position on the screen) is defined as:

**Definition 14.2** (*screenOutput* of Verisoft Email Client)**.** *Let $s$ be a list of strings representing the email message. Let strings by lists of datatype `char`, and let `blank` $\in$ `char` be the blank. Let $co$ be the column offset, and let $ro$ be the row*

*offset. Let $w$ be the width of the screen and $h$ be the height of the screen. Then function*

$$\textit{screenOutput} : (x, y) \rightarrow \text{char} \quad \textit{with } 0 \leq x < w \textit{ and } 0 \leq y < h$$

*is defined as*

$$\textit{screenOutput}[y, x] = \begin{cases} s[y + ro][x + co] & \textit{if length}(s) < y + ro \textit{ and} \\ & \quad \textit{length}(s[y + ro]) < x + co \\ \textit{blank} & \textit{otherwise} \end{cases}$$

| | |
|---|---|
| **context** | `execute(cmd, m)` |
| **pre** | `cmd` $\in$ { `CURSOR_LEFT`, `CURSOR_RIGHT`, `CURSOR_UP`, `CURSOR_DOWN`, `INSERT_CHAR`, `DELETE_CHAR`, `QUIT` } |
| **post** | **if** `cmd` = `CURSOR_LEFT` **then** |
| | *cursorLeftPostcondition* **and** |
| | result = `CURSOR_MOVED` |
| | **else if** `cmd` = `CURSOR_RIGHT` **then** |
| | *cursorRightPostcondition* **and** |
| | result = `CURSOR_MOVED` |
| | **else if** `cmd` = `CURSOR_UP` **then** |
| | *cursorUpPostcondition* **and** |
| | result = `CURSOR_MOVED` |
| | **else if** `cmd` = `CURSOR_DOWN` **then** |
| | *cursorDownPostcondition* **and** |
| | result = `CURSOR_MOVED` |
| | **else if** `cmd` $\in$ `INSERT_CHAR` **then** |
| | *insertCharPostcondition* **and** |
| | result = `CHAR_INSERTED` |
| | **else if** `cmd` = `DELETE_CHAR` **then** |
| | *deleteCharPostcondition* **and** |
| | result = `CHAR_DELETED` |
| | **else** |
| | result = `QUIT` |
| | **end if** |

Table 14.2: Command execution function

While the main execution loop is generic, the specification of `execute` depends on the actual implementation. The part of the Verisoft email client's spec-

ification of `execute` relevant to editing email is defined by the OCL specification given in Table 14.2. *execute* takes a command (`cmd`) and a `m`, a representation of the email as input. For the commands `CURSOR_LEFT`, `CURSOR_RIGHT`, `CURSOR_UP`,`CURSOR_DOWN`, `INSERT_CHAR`, `DELETE_CHAR`, it guarantees that value of `m` after execution represents the application by *insertCharPostcondition*, *deleteCharPostcondition*, *cursorLeftPostcondition*, *cursorRightPostcondition*, *cursorUpPostcondition*, *cursorDownPostcondition*,respectively. These definitions describe the desired results of the respective operations.

As an example, we provide a definition for *cursorRightPostcondition.* We will come back to this in Chapter 16. The reader is referred to Chapter 16 for an for definitions of auxillary function *flatCPos*.

**Definition 14.3** (*cursorRightPostcondition*)**.**

$$\begin{aligned}&\mathit{cursorRightPostcondition} \equiv \\ &\mathit{flatCPos}(\texttt{m.st}, \texttt{m.cx}, \texttt{m.cy}) = \\ &\mathit{flatCPos}(\texttt{m.st@pre}, \texttt{m.cx@pre}, \texttt{m.cy@pre}) + 1\end{aligned}$$

# Chapter 15

# Authentication and Secure Channels

## 15.1 Confidentiality

In this chapter, we show how the definitions and theorems from Chapter 11 relating to Confidentiality are used to specify confidentiality requirement for an email client.

According to the system model developed in Chapter 9, the email system is a multi users, multi application system. Without loss of generality, we assume the legitimate user of the email system is $u_0$, and the legitimate email application is $a_0$. We further assume there are exactly two devices, the keyboard identified as $d_0$, and the screen identified as $d_1$.

In Chapter 11 confidentiality of a channel Channel $(s, d, r)$ in a system model *system*$(U, A, I, O, M, u, a, \lambda)$ has been defined (Definition 11.1) as

$$\begin{array}{l}\textit{Confidentiality} \equiv \\ \forall\, s, d, r, d', r', m, m'. \\ [c_{(s,d)}\ \textbf{xmt}\ m] \wedge \textbf{EF}[c_{(d',r')}\ \textbf{xmt}\ m'] \wedge m \sim m' \wedge \textit{secret}(s, d', r', m') \rightarrow \\ \textit{legitimate}(s, d', r', m')\end{array}$$

Confidentiality is guaranteed if all secret messages which may be eventually received (**EF**$[c_{(d',r')}$ **xmt** $m'] \wedge$ *secret*$(s, d', r', m')$) have been sent ($[c_{(s,d)}$ **xmt** $m] \wedge m \sim m'$) by a legitimate sender (*legitimate*$(s, d', r', m')$).

Furthermore, we showed that confidentiality is satisfied when secret messages are sent only on trusted and legitimate paths to legitimate recipients, defined as *ConfCond* (Definition 11.4):

$$\begin{array}{lll}\textit{ConfCond} \equiv \forall\, s, d, r, m. & [c_{(s,d)}\ \textbf{xmt}\ m] \wedge \textit{secret}(s, d, r, m) \rightarrow \\ & \textit{legitimate}(s, d, r, m) \wedge \textit{trusted}((s, d, r))\end{array}$$

The confidentiality condition defines that secret messages are sent to legitimate receivers on trusted paths only. We have shown that the confidentiality condition

is satisfied if both the user and the application are attentive (Theorem 11.2), with being attentive defined in Definition 11.5 as

$$
\begin{aligned}
&\textit{AttentiveParty}(s) \equiv \\
&\forall\, d, r, m.[c_{(s,d)}\ \textbf{xmt}\ m] \wedge \textit{secret}(s, d, r, m) \rightarrow \\
&\qquad \wedge\ (\textit{asmModifies}(s, (s, d, r)) \leftrightarrow \textit{modifies}((s, d, r))) \\
&\qquad \wedge\ (\textit{asmPrivate}(s, (s, d, r)) \leftrightarrow \textit{private}((s, d, r))) \\
&\qquad \wedge\ \textit{asmIdentitiy}(s, (s, d, r), s, d, r) \\
&\qquad \wedge\ \neg\textit{asmModifies}(s, (s, d, r)) \\
&\qquad \wedge\ \textit{asmPrivate}(s, (s, d, r)) \\
&\qquad \wedge\ \textit{legitimate}(s, d, r, m)
\end{aligned}
$$

Intuitively, attentiveness means that secret messages are send only if

- it is legitimate to send them,
- the assumptions about the channel in respect to modification of messages, privacy of the channel, and identity of the communicating party are correct,
- and the channel is private and not modifying data.

In the email application, there are two channels: One user input channel, connecting the user via the keyboard to the application, and one application output channel, connecting the application to the user via the screen. We assume that all communications on both of these channels are both *secret* and *legitimate*, i.e. the user and the application are free to share all information with each other, but they are not allowed to share information with anybody else.

**Definition 15.1** (Email System Model)**.** *Let* $S = \textit{system}(U, A, I, O, M, u, a, \lambda)$ *be a system model.* $E = \textit{emailSystem}(U, A, I, O, M, u, a, \lambda, k, t)$ *with* $k \in I$*,* $t \in O$*, and*

$$
\begin{aligned}
&\forall\, s, d, r, m.\mathit{secret}(s, d, r, m) \\
&\forall\, m.\mathit{legitimate}(u, k, a, m) \\
&\wedge\ \forall\, m.\mathit{legitimate}(a, t, u, m) \\
&\wedge\ \forall\, r, m.r \neq a \vee d \neq k \rightarrow \neg\mathit{legitimate}(u, d, r, m) \\
&\wedge\ \forall\, s, m.t \neq u \vee d \neq t \rightarrow \neg\mathit{legitimate}(t, d, a, m)
\end{aligned}
$$

*is an email system model. In this model,* $k$ *represents the keyboard and* $t$ *represents the screen.*

Since we model HCI only, sending and receiving email on the network is not subject of the definitions of *secret* and *legitimate*.

From the definition of *AttentiveParty* it follows trivially (and has been shown in Proof 11.2) that whenever a message is send by the user or the application,

the path has to be private, the message must not be modified on the path, and the receiver of the message is correct. Additionally the sender must always know when these conditions hold. This leads to the following requirement specifications for screen and keyboard:

**Definition 15.2** (Confidentiality Requirement Specification Screen)**.** *Let $E = emailSystem(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model. Screen component $t$ satisfies the confidentiality requirement if*

$$E \models \mathbf{AG}(\neg modifies((a, t, u)) \wedge private((a, t, u)))$$

**Definition 15.3** (Confidentiality Requirement Specifications Keyboard)**.** *Let $E = emailSystem(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model. Keyboard component $k$ satisfies the confidentiality requirement if*

$$E \models \mathbf{AG}(\neg modifies((u, k, a)) \wedge private((u, k, a))$$

If screen and keyboard satisfy their confidentiality requirements, the user and the application must know that the channel is private and does not modify messages, and that they are sending on the right channel. The user will use the keyboard to enter messages and no other input device (**AG**$(\forall m.[c_{(u,d)}$ **xmt** $m] \rightarrow d = k)$), and he assumes that

- the keyboard device is private:
  $asmPrivate(u, (u, k, a))$,
- the keyboard device is not modifying messages:
  $\neg asmModifies(u, (u, k, a))$
- and the correct application is listening to the keyboard device:
  $asmIdentitiy(u, (u, k, a), u, k, a)$

**Definition 15.4** (Confidentiality Requirement Specification for User)**.** *Let $E = emailSystem(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model. User $u$ satisfies the confidentiality requirement if*

$$\begin{aligned} E \models & \mathbf{AG}(\forall m.[c_{(u,d)}\ \mathbf{xmt}\ m] \rightarrow d = k) \wedge \\ & \mathbf{AG}(\neg asmModifies(u, (u, k, a)) \wedge asmPrivate(u, (u, k, a)) \\ & \quad asmIdentitiy(u, (u, k, a), u, k, a)) \end{aligned}$$

The confidentiality requirement definition for the application is similar to the requirement definition for the user, with the device in question being the screen rather than the keyboard:

**Definition 15.5** (Confidentiality Requirement Specification for Application)**.** *Let* $E = \mathit{emailSystem}(U, A, I, O, M, u, a, \lambda, k, t)$ *be an email system model. Application* $a$ *satisfies the confidentiality requirement if*

$$E \models \mathbf{AG}(\forall m.[c_{(a,d)}\ \mathbf{xmt}\ m] \rightarrow d = t) \wedge \mathbf{AG}(\neg \mathit{asmModifies}(a,(a,t,u)) \wedge \mathit{asmPrivate}(a,(a,t,u)) \mathit{asmIdentitiy}(a,(a,t,u),a,t,u))$$

Next, we show that a system guaranteeing the component specifications for screen, keyboard, user, and application is confidential.

**Theorem 15.1.** *Let* $E = \mathit{emailSystem}(U, A, I, O, M, u, a, \lambda, k, t)$ *be an email system model satisfying definitions 15.2 to 15.5. Then the following holds:*

$$E \models \mathit{ConfCond}$$

**Proof**

Insert definition of *ConfCond*:

$$E \models \forall s, d, r, m.\quad [c_{(s,d)}\ \mathbf{xmt}\ m] \wedge \mathit{secret}(s,d,r,m) \rightarrow \mathit{legitimate}(s,d,r,m) \wedge \mathit{trusted}((s,d,r))$$

*ConfCond* holds trivially unless messages are transmitted. For message transmission (i.e. $[c_{(s,d)}\ \mathbf{xmt}\ m]$), we distinguish three cases:

1. $s \neq u$ and $s \neq a$
2. $s = u$
3. $s = a$

**First case:** $s \neq u$ **and** $s \neq a$ From the definition of the email system model (Definition 15.1), it follows that $\mathit{secret}(s,d,r,m)$ is never true. Therefore the theorem trivially holds.

**Second case:** $s = u$

$$E \models \forall d, r, m.\quad [c_{(u,d)}\ \mathbf{xmt}\ m] \wedge \mathit{secret}(u,d,r,m) \rightarrow \mathit{legitimate}(u,d,r,m) \wedge \mathit{trusted}((u,d,r))$$

From the confidentiality requirement specification for the user (Definition 15.4) it follows that the user sends messages to the keyboard only. Therefore

$$E \models \forall r, m.\quad [c_{(u,k)}\ \mathbf{xmt}\ m] \wedge \mathit{secret}(u,k,r,m) \rightarrow \mathit{legitimate}(u,k,r,m) \wedge \mathit{trusted}((u,k,r))$$

From Definition 15.4 it also follows that the user knows about the identity of the receiving party. Therefore

$$E \models \forall\, m.\quad [c_{(u,k)}\ \textbf{xmt}\ m] \wedge \textit{secret}(u,k,a,m) \rightarrow \\ \textit{legitimate}(u,k,a,m) \wedge \textit{trusted}((u,k,a))$$

$\forall\, m.\textit{secret}(u,k,a,m) \rightarrow \textit{legitimate}(u,k,a,m)$ follows from the email system definition. It remains to be shown that

$$E \models \textit{trusted}((u,k,a)$$

Insert definition of trusted:

$$E \models \textit{authenticated}((u,k,a)) \wedge \neg\textit{leaks}((u,k,a)) \\ \wedge \neg\textit{modifies}((u,k,a))$$

With $\textit{private}((u,k,a)) \rightarrow \neg\textit{leaks}((u,k,a))$ and the keyboard requirement specification (Definition 15.3), it remains to be shown that

$$E \models \textit{authenticated}((u,k,a))$$

Insert definition of authenticated (Definition 10.4):

$$E \models \textit{asmIdentity}(s,(s,d,r),s,d,r) \\ \wedge \textit{asmIdentity}(t,(s,d,r),s,d,r)$$

This follows directly from the requirement specifications for the user and the application.

**Third case** The third case is proven in the same way as the second case. □

## 15.2 Authenticity

We have shown that confidentiality is guaranteed if the keyboard device, the screen device, the application, and the user satisfy the confidentiality requirement (Definitions 15.3 to 15.5.) For the keyboard and the screen, the only requirements are that they are private and do not modify the content of the message. Measures to ensure these requirements are partly technical and under the responsibility of the operating system, and partly organizational. For example, the workspace must be set up in such a way that no non-authorized party can read the screen. The

requirements for the user and application components require that the user and the application do know about the identity of the other party. On the application's side, guaranteeing authenticity of the user is under the responsibility of the operating system, which uses an authenticity procedure as described in Common Criteria Class FIA (Identification and Authentication), which has been formalized in Section 10.2.

It is not sufficient that the authenticity of the user is guaranteed. The authenticity of the application must be guaranteed as well. Since screen output is the only input source for the user, the information about the authenticity of the application must be shown on the screen. In Section 14.3, a screen output function specified in OCL has been developed. In the following, the specification is adapted to carry the information needed to identify the application.

Locking a resource is not sufficient to guarantee security. The user must also *know* which process locks a resource and whether the system is busy or not. Therefore, the operating system configuration must be shown to the user represented by a string of characters. We assume this string representation to be given by the function

$$\mathit{OSConfString} : \texttt{OSConf} \rightarrow \texttt{String} \ ,$$

which we do not further specify here. It must return a string that allows the user to determine the exact operating system configuration. Its actual implementation depends, for example, on the language(s) the user is supposed to understand.

| |
|---|
| **context** `updateScreen(status, flag, cmdResult)` |
| **post** *stringAt*$(0,1)$ = *stateAsString*(`status`) and |
| **if** `flag = waiting` **then** |
| *stringAt*$(0,2)$ = "Ready" |
| **else** |
| *stringAt*$(0,2)$ = "Processing" |
| **end if** |
| and |
| $\forall\, k \in \{3,\ldots,$ *screenHeight* $-3\}$ : |
| *stringAt*$(0,k)$ = |
| *screenOutput*(`cmdResult`)$[k-3]$ |

Table 15.1: Refined specification of the application's function for updating the screen contents, taking into account that the first line is under control of the operating system.

We assume that the first line of the screen is reserved for information on the operating system configuration, i.e. the first line should be identical to *OSConf-*

*String*(`OSConf`). By not allowing processes (other than the operating system) access to row 0, the changed specifications of `setChar` and `setCursor` given below assure that the configuration information can be overwritten neither by any user applications nor by any attacking processes.

We specify the correct display of the operating configuration resources as an invariant of `OSConf`:

| | |
|---|---|
| **context** | `OSConf` |
| **inv** | $stringAt(t)[0,0] = \mathit{OSConfString}(\texttt{OSConf})$ |

Note that the operating system invariant guaranteeing that the application name is shown in the status line is not part of the specification of `updateScreen`, because it is under the responsibility of the operating system to guarantee this property. If the status line were under control of the application, a malicious program could write wrong information in the line, making users believe they are interacting with a different application. The specification of `updateScreen` given in Table 14.1 does not take into account that the first line of the screen is under control of the operating system. A refined specification is given in Table 15.1.

# Chapter 16

# Availability

In Chapter 11, the common definition of availability as reachability of desirable states and avoidability of undesirable states has been adapted to user interface security. Availability is guaranteed if for a given user and application model, desirable states are always reached and undesirable states are never reached. This led to Definition 11.3:

$$\mathit{Availability} \equiv \mathbf{AG}(\neg \mathit{fatal} \rightarrow \mathbf{AF}\mathit{success})$$

In this Chapter, we use this definition to define availability requirements for a secure email client. We specify concrete application and user components and show that a model constructed from the components satsifies the availability requirement. Based on the application logic component specification, a pervasively verified email client has been developed in project Verisoft. The concrete component definitions developed in this chapter are specific to the secure email client developed in Verisoft. We present it as an example for an application of the methodology developed in Parts I and II.

Figure 16.1 gives a state chart model of the email client. The email client has two functionalities: First, it should be possible to write, sign, and send email. Second, it should be possible to receive email, check a signature, and read the email. In the following we provide suitable user models for both scenarios.

## 16.1 Writing, Signing, Sending Email

In this scenario, the user first writes an email, then signs it, and finally sends it. The proof of availability of this functionality is split into two parts. One part of the proofs uses model checking to show that certain abstract states are reached. This part proves the temporal aspects of the theorem based on the abstract state chart model of the email client. The second part of the proof uses Hoare logic to

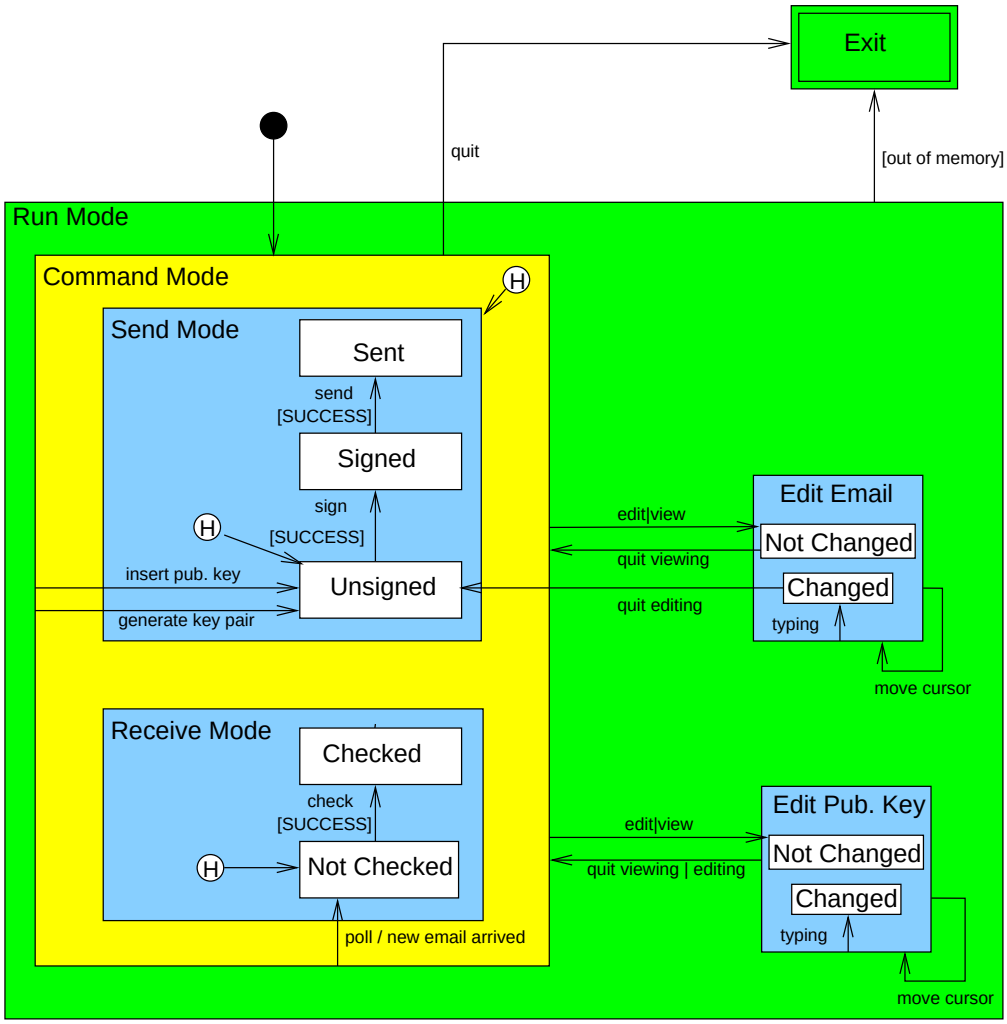


Figure 16.1: Statechart of email client applications. State transitions represent execution of program functions.

prove properties of the actual state transitions. In order to proof the theorem, both parts are needed: The model checking based proofs allow to express properties like “state ‘sent’ is always reachable”, or “state ‘email changed’ can be reached an arbitrary number of times.” However, since these proofs operate on an abstract state model of the application, and not on the actual states, it is not possible to show that “an arbitrary email can be entered”, because the actual email is not part of the model. Properties of the actual state space of the system are proven in Hoare logic.

### 16.1.1 CTL Part

```
GOAL: WRITE_SIGN_SEND
    GOAL: WRITE
        OPERATOR: EDIT_MAIL
        GOAL: MOVE_TO_BEGIN_OF_MAIL
            OPERATOR: MOVE_CURSOR_LEFT
            OPERATOR: MOVE_CURSOR_LEFT
            ...
        GOAL: DELETE_ALL_CHARACTERS
            OPERATOR: DELETE_CHAR
            OPERATOR: DELETE_CHAR
            ...
        GOAL: TYPE_MAIL
            OPERATOR: INSERT_CHAR
            OPERATOR: INSERT_CHAR
            ...
    OPERATOR: SIGN
    OPERATOR: SEND
```

Figure 16.2: GOMS model for writing, signing, sending mail. Repeated operations are indicated by “...”.

Figure 16.2 gives a GOMS model of a user writing, signing, and sending email. The user first moves to the begin of the mail. Then he deletes all the old content of the mail. Finally, he types the intended message, signs and send it. An IOLTS of the GOMS model[1] is given in Figure 16.3. The email system should guarantee that it is always possible to execute these steps, resulting in the

[1] Some states have been joined; see Definition 6.1.

sending of the intended message, signed by the sender. Furthermore, it should not be possible to send a message without signing it. Formal definitions of these concepts are given in the following theorems.

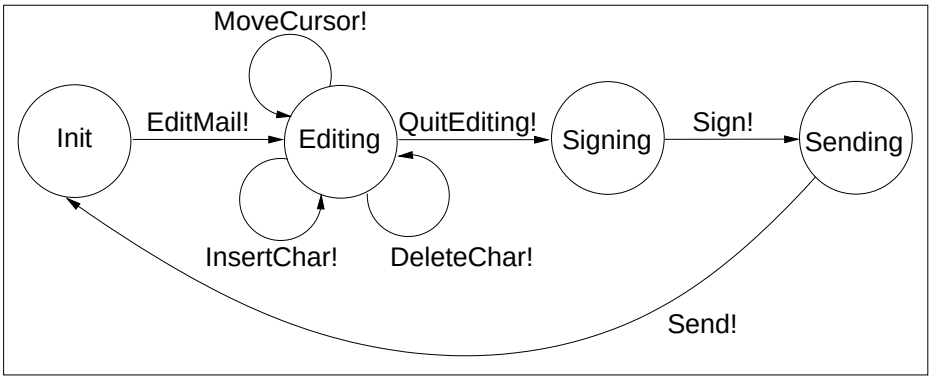


Figure 16.3: User Model for Writing, Signing, and Sending Email

The first theorem states that it is possible to enter edit mode and move the cursor an arbitrary number of times while staying in state `mailNotChanged`:

**Theorem 16.1** (Arbitrary number of cursor moves)**.**
*Let $E = emailSystem(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model with $a$ corresponding to the IOLTS given in Figure 16.1 and $u$ corresponding to the IOLTS given in Figure 16.3. Then the following holds:*

$$\boldsymbol{EF}(\boldsymbol{EG}((\mathit{user.action} = \text{moveCursor}) \wedge (\mathit{client.state} = \text{mailNotChanged})))$$

In combination with the Hoare specification from Section 16.1.2, it follows that it is always possible to reach the beginning of the email by repeatedly issuing command `cursorLeft`.

The next theorem states that if the client is in state `mailNotChanged`, it is possible to transit to state `mailChanged` in the next step:

**Theorem 16.2** (Transition from `mailNotChanged` to `mailChanged`)**.**
*Let $E = emailSystem(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model with $a$ corresponding to the IOLTS given in Figure 16.1 and $u$ corresponding to the IOLTS given in Figure 16.3. Then the following holds:*

$$\begin{aligned}&\boldsymbol{EF}((\mathit{client.state} = \texttt{mailNotChanged}) \\ &\quad \wedge \boldsymbol{EX}(\mathit{client.state} = \texttt{mailChanged}))\end{aligned}$$

If the client is in state mailChanged, an arbitrary number of characters can be deleted:

**Theorem 16.3** (Deleting an arbitrary number of characters)**.**
*Let $E = emailSystem(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model with $a$ corresponding to the IOLTS given in Figure 16.1 and $u$ corresponding to the IOLTS given in Figure 16.3. Then the following holds:*

$$\boldsymbol{EF}(\boldsymbol{EG}((user.action = \texttt{deleteChar}) \\ \wedge (client.state = \texttt{mailChanged})))$$

In combination with the Hoare logic proofs from Section 16.1.2, it follows from Theorems 16.1 to 16.3 that the user can get the application in a state where the email message is empty.

The next theorem states that an arbitrary number of characters can be inserted:

**Theorem 16.4** (Inserting arbitrary characters)**.**
*Let $E = emailSystem(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model with $a$ corresponding to the IOLTS given in Figure 16.1 and $u$ corresponding to the IOLTS given in Figure 16.3. Then the following holds:*

$$\boldsymbol{EF}(\boldsymbol{EG}((user.action = \texttt{insertChar}) \\ \wedge (client.state = \texttt{mailChanged})))$$

In combination with Hoare logic proof about inserting characters from Section 16.1.2, it follows that the user can enter an arbitrary email into the application.

The last two theorems ensure that sending the email is possible, and that no unsigned messages are send:

**Theorem 16.5** (Sending Possible)**.**
*Let $E = emailSystem(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model with $a$ corresponding to the IOLTS given in Figure 16.1 and $u$ corresponding to the IOLTS given in Figure 16.3. Then the following holds:*

$$\boldsymbol{EF}((client.state = \texttt{mailChanged}) \\ \wedge \boldsymbol{EX}(\boldsymbol{E}[(client.state != \texttt{mailChanged})\boldsymbol{U}(client.state = \texttt{sent})]))$$

**Theorem 16.6** (No Unsigned Messages Sent)**.**
*Let $E = emailSystem(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model with $a$ corresponding to the IOLTS given in Figure 16.1 and $u$ corresponding to the IOLTS given in Figure 16.3. Then the following holds:*

$$\boldsymbol{AG}((\boldsymbol{EX}(client.state = \texttt{sent})) \rightarrow \\ (client.state = \texttt{signed}))$$

**Proof**

Appendix E gives the conjunction of theorems 16.1 to 16.6 in the input format of theNuSMV (Cimatti et al., 2002) model checker. Correctness has been proven in NuSMV 2.3.0. □

### 16.1.2 Hoare Part

We want so show that the user can enter an arbitrary email. We show this in three steps: First, we show that the user can always move the cursor to the beginning of the current email message by repeatedly giving command `cursorLeft`. Second, we show that the user can delete the current email message if he is at the begin of the message and gives command `deleteChar` repeatedly. Finally, we show that the user can create an arbitrary text by typing the characters of the text successively, starting with an empty email.

**The user can always reach the beginning of a message by repeatedly issuing command `cursorLeft`.** In Section 16.1.1, we have shown that the user can execute command `cursorLeft` an arbitrary number of times. Now, we show that the user will reach the beginning of the email eventually if he gives command cursorLeft repeatedly. A Hoare specification[2] of command `cursorLeft` is given in Figure 16.4. `m` is the data structure storing the email text as a list of strings, where strings are lists of characters. The list of strings is accessed via `m.st`. The cursor x and y positions are accessed via `m.cx`, `mail.cy`, respectively. The procedure specification of `cursorLeft`, `cursorRight`, `insertChar`, and `deleteChar` make use of auxiliary functions *flatCPos*, *flatten*, *flattenUntilPos*, and *flattenFromPos*, which themselves use auxiliary functions *take* and *drop*. These functions are defined in Definitions 16.1 to 16.6. Function *flatten* flattens a two dimensional string. Function *flattenUntilPos* flattens a two dimensional string up to a given $(x, y)$ position. Function *flattenFromPos* flattens a two dimensional string starting at a given $(x, y)$ position. Function *take* takes the first characters of a string, and *drop* removes the first characters from a string. Function *flatCPos* translates a two-dimensional cursor position into the position of the cursor in a one-dimensional string. In the definition of these functions, we use the function *head*, which gives the first element of a list, *tail*, which gives the last element of a list, and *concat*, which concatenates two or more lists.

**Definition 16.1** (*flatten*)**.** *Let $s$ be a list of strings. flatten is defined as*

$$flatten(s) \quad = \quad \begin{cases} "" & \textit{If} |s| = 0 \\ concat([head(s)], flatten(rest(s)) & otherwise \end{cases}$$

**Definition 16.2** (*flattenUntilPos*)**.** *Let $s$ be a list of strings, and let $x$ and $y$ be natural numbers such that the position $(x, y)$ exists in list of strings $s$, i.e. $0 \leq y <$*

[2]The Hoare specifications given here for `cursorLeft`, `cursorRight`, `insertChar`, and `deleteChar` are somewhat simplified, because the actual data structures of the email client are more complex than the data structures shown here. See Beckert et al. (2007) for the actual specifications.

$|s|$ *and* $0 \leq x < |s[y]|$. *flattenUntilPos is defined as*

$$flattenUntilPos(x, y, s) = \begin{cases} \textit{take}(x, head(s)) & \textit{If } y = 0 \\ flatten(head(s), & \textit{Otherwise} \\ \quad flattenUntilPos(x, y-1, tail(s))) & \end{cases}$$

**Definition 16.3** (*take*)**.** *Let* $l$ *be a list, and let* $x$ *be a position in list* $l$*, i.e.* $0 \leq x < |l|$ *take is defined as*

$$\textit{take}(x, l) = \begin{cases} \text{""} & \textit{If } x = 0 \\ flatten(head(l), \textit{take}(x-1, tail(l))) & \textit{Otherwise} \end{cases}$$

**Definition 16.4** (*flattenFromPos*)**.** *Let* $s$ *be a list of strings, and let* $x$ *and* $y$ *be natural numbers such that the position* $(x, y)$ *exists in list of strings* $s$*, i.e.* $0 \leq y < |s|$ *and* $0 \leq x < |s[y]|$. *flattenFromPos is defined as*

$$flattenFromPos(x, y, s) = \begin{cases} flatten(\textit{drop}(x, head(s)), tail(s)) & \textit{If } y = 0 \\ flattenFromPos(x, y-1, tail(s)) & \textit{Otherwise} \end{cases}$$

**Definition 16.5** (*drop*)**.** *Let* $l$ *be a list, and let* $x$ *be a position in list* $l$*, i.e.* $0 \leq x < |l|$ *drop is defined as*

$$\textit{drop}(x, l) = \begin{cases} l & \textit{If } x = 0 \\ \textit{drop}(x-1, tail(l)) & \textit{Otherwise} \end{cases}$$

**Definition 16.6** (*flatCPos*)**.** *Let* $s$ *be a list of strings, and let* $x$ *and* $y$ *be natural numbers such that the position* $(x, y)$ *exists in list of strings* $s$*, i.e.* $0 \leq y < |s|$ *and* $0 \leq x < |s[y]|$. *flatCPos is defined as*

$$\textit{flatCPos}(s, x, y) \quad = \quad \begin{cases} x & \textit{If } y = 0 \\ \textit{flatCPos}(\textit{tail}(s), x, y-1) & otherwise \\ \quad +\textit{len}(\textit{head}(m)) & \end{cases}$$

With these auxiliary functions, procedure `cursorLeft` of the email client can be specified. The specification uses function *flatCPos* to translate the position of the cursor in the two-dimensional list of strings into the same position in a flat,

**context** `cursorLeft(m)`
**post** **if** *flatCPos*(`m.st`,`m.cx`,`m.cy`) $>$ 0 **then**
*flatCPos*(`m.st`,`m.cx`,`m.cy`) $=$
*flatCPos*(`m.st@pre`,`m.cx@pre`,`m.cy@pre`) $-$ 1
result = RESULT_CURSOR_MOVED
**else**
result = RESULT_MOVE_CURSOR_FAILED
**end if**

Figure 16.4: Specification of `cursorLeft`

one-dimensional string. It ensures that after execution of `cursorLeft`, the location of the cursor in the flat string is decreased by one. The specification of `cursorLeft` is given in Figure 16.4. In the specification of `cursorLeft`, `m` is a data structure representing the current configuration of the mail editing component. `m.st` is a list of strings representing the email, and `m.cx` and `m.cy` are the $(x, y)$ positions of the cursor. Reachability of position $(0,0)$ by repeatedly executing command `cursorLeft` follows trivially from the specification.

**context** `deleteChar()`
**post** *flatten*(`m.st`) $=$
*concat*(*flattenUntilPos*(`m.cx`,`m.cy`,`m.st@pre`),
*tail*(*flattenFromPos*(`m.cx`,`m.cy`,`m.st@pre`)))

Figure 16.5: Specification of `deleteChar`

Definition 16.5 specifies `deleteChar`. Like `cursorLeft`, `deleteChar` is specified on the flat representation of the list of strings provided by *flatten*. It ensures that the flat representation of the list of strings after execution of the procedure is identical to the flat representation of the original list of strings up to the current cursor position, concatenated to the flat representation of the list of strings from the current cursor position with the first character removed.

Next, we show that the email message can be deleted under the assumption that the cursor is at position $(0,0)$ and the user gives command `deleteChar` repeatedly.

**Theorem 16.7.**
*Let $E =$ emailSystem$(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model with $a$ corresponding to the IOLTS given in Figure 16.1 and `deleteChar` as specified in Figure 16.5. Let `m.st` be the list of strings representing the email, `m.cx` the $x$ position of the cursor and `m.cy` the $y$ position with `m.cx` $= 0$ and `m.cy` $= 0$*

*Then*

$$|m.st| = 0$$

*if* `deleteChar` *is executed sufficiently often.*

**Proof**

We proof the proposition by induction over the length of the message.

**Base case:** $|\texttt{m.st}| = 0$
In the base case, the mail is already empty

**Induction step:**
We know that $|\texttt{m.st}| = n+1 \wedge \texttt{m.x} = 0 \wedge \texttt{m.y} = 0$ and that the theorem holds for $|\texttt{m.st}| = n$. Since the cursor is at position $(0,0)$, the following equations are true:

$$\begin{aligned} \mathit{flatten}(\textit{flattenUntilPos}(\texttt{m.cx},\texttt{m.cy},\texttt{m.st@pre}) &= [] \\ \mathit{flatten}(\textit{flattenFromPos}(\texttt{m.cx},\texttt{m.cy},\texttt{m.st@pre}) &= \textit{m.st@pre} \end{aligned}$$

With the specification of `deleteChar` it follows that $\texttt{m.st} = \mathit{tail}(\texttt{m.st@pre})$. From the definition of *tail* it follows that $|\textit{m.st}| = \mathit{tail}(\textit{m.st@pre}) - 1$ and finally $|\textit{m.st}| = n$. □

**context** `insertChar(char c)`
**post** $\mathit{flatten}(\texttt{m.st}) =$
$\mathit{concat}(\textit{flattenUntilPos}(\texttt{m.cx},\texttt{m.cy},\texttt{m.@pre}), c,$
$\textit{flattenFromPos}(\texttt{m.cx},\texttt{m.cy},\texttt{m.st.row@pre}))$
**and** $\mathit{flatCPos}(\texttt{m.st},\texttt{m.cx},\texttt{m.cy}) =$
$\mathit{flatCPos}(\texttt{m.st@pre},\texttt{m.cx},\texttt{m.cy}) + 1$

Figure 16.6: Specification of `insertChar`

Finally, we show that the user can enter an arbitrary email message by inserting characters if the message is empty. The specification of `insertChar` given in Figure 16.6 is very similar to the specification of `deleteChar`. It ensures that the resulting flat string is identical to the concatenation of original flat string up to the cursor position, the new character, and the original flat string from the cursor position to the end. Additionally the cursor has been moved one character to the right.

**Theorem 16.8.**
*Let $E = \mathit{emailSystem}(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model with $a$ corresponding to the IOLTS given in Figure 16.1 and `insertChar` as specified in Figure 16.6. Let `m.st` be the list of strings representing the email with `m.st` $= []$, `m.cx` the $x$ position of the cursor and `m.cy` the $y$ position with `m.cx` $= 0$ and `m.cy` $= 0$ Let message be the message the user wants to enter.*

*Then*

$$\mathit{flatten}(\texttt{m.st}) = \mathit{message}$$

*if `insertChar` is executed sufficiently often.*

**Proof**

The theorem is proven by induction over the length of the messages, with the induction hypothesis that after entering $n$ characters, the first $n$ characters of the message and the mail are identical and the cursor is at the last position of the mail.

**Base case:** $n = 0$
$\mathit{take}(\mathit{message}, n) = [] = \mathit{flatten}(\texttt{m.st})$
and furthermore $\mathit{flatCPos}(\texttt{m.st}, \texttt{m.cx}, \texttt{m.cy}) = |\mathit{flatten}(\texttt{m.st})|$

**Induction step**
From the induction hypothesis we know that
$\mathit{take}(\mathit{message}, n) = \mathit{flatten}(\texttt{m.st@pre})$
and $\mathit{flatCPos}(\texttt{m.st@pre}, \texttt{m.cx@pre}, \texttt{m.cy@pre}) = n + 1$
From
$\mathit{flatCPos}(\texttt{m.st@pre}, \texttt{m.cx@pre}, \texttt{m.cy@pre}) = \mathit{flatten}(\texttt{m.st@pre}) + 1$
It follows that
$\mathit{flattenUntilPos}(\texttt{m.cx}, \texttt{m.cy}, \texttt{m.@pre}) = \mathit{flatten}(\texttt{m.st@pre})$
and $\mathit{flattenFromPos}(\texttt{m.cx}, \texttt{m.cy}, \texttt{m.@pre}) = []$.
With the specification of `insertChart` it follows that

$$\begin{aligned} \texttt{m.st} &= \mathit{flatten}(\mathit{flatten}(\texttt{m.st@pre}), \mathit{message}[n+1]) \\ &= \mathit{take}(\mathit{message}, n+1) \end{aligned}$$

and

$$\begin{aligned} \mathit{flatCPos}(\texttt{m.st}, \texttt{m.cx}, \texttt{m.cy}) &= \mathit{len}(\mathit{flatten}(\texttt{m.st@pre}) + 1 \\ &= \mathit{flatten}(\texttt{m.st}) \end{aligned}$$

□

## 16.2 Receiving, Checking, Reading Email

### 16.2.1 CTL Part

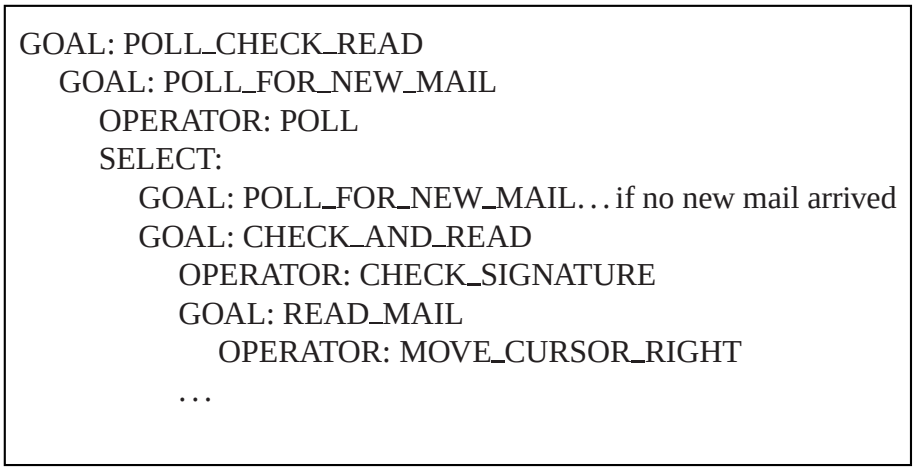


Figure 16.7: GOMS model for polling email, checking asignature, and reading a mail. Repeated operations are indicated by "...".

The theorems and proofs in this section are closely related to the theorems from Section 16.1. Again, we start with the CTL part. Figure 16.7 gives a GOMS model of a user polling for new email, checking the signature of a newly arrived email, and reading it. The user polls email until new email arrives. Then he checks the signature. After checking the signature, he reads the mail. Here, we define reading mail as "moving the cursor over the email text." An IOLTS of the GOMS model[3], is given in Figure 16.8.

We want to guarantee that email is received eventually if the user repetitively polls for new mail. If new mail arrived, the user should be able to read the mail, i.e. it should be possible to move the cursor over all characters of the email. It should also be guaranteed that the user checks the signature of newly arrived mail. These requirements are formalized in the following theorems:

If the system is in the initial state and new mail is available, then both the new mail will have arrived in the next step and the user will know about.

**Theorem 16.9** (Successful Polling)**.**
*Let* $E = \mathit{emailSystem}(U, A, I, O, M, u, a, \lambda, k, t)$ *be an email system model with* $a$ *corresponding to the IOLTS given in Figure 16.1 and* $u$ *corresponding to the*

[3]Some states have been joined; see Definition 6.1.

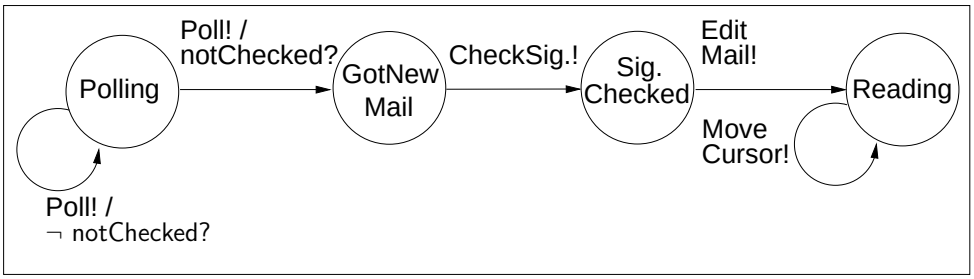


Figure 16.8: User Model for Receiving Email, Checking the Signature, and Reading the Email

*IOLTS given in Figure 16.8. Then the following holds:*

$$(\boldsymbol{AG}(((\textit{client.state} = \text{unsigned}) \wedge \textit{client.newMailAvailable}) \rightarrow \\ \boldsymbol{AX}((\textit{client.state} = \text{notChecked}) \wedge \boldsymbol{AX}(\textit{user.state} = \text{got\_new\_mail}))))$$

If the system is in the initial state and no mail is available, then both the system and the user will remain in their initial states in the next step.

**Theorem 16.10** (Unsuccessful Polling)**.**
*Let $E = \textit{emailSystem}(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model with $a$ corresponding to the IOLTS given in Figure 16.1 and $u$ corresponding to the IOLTS given in Figure 16.8. Then the following holds:*

$$(\boldsymbol{AG}(((\textit{client.state} = \text{unsigned}) \wedge \neg\textit{client.newMailAvailable}) \rightarrow \\ \boldsymbol{AX}((\textit{client.state} = \text{unsigned}) \wedge \boldsymbol{AX}(\textit{user.state} = \text{end}_\text{p}\text{olling}))))$$

The user will continue polling for new mail while the system is in its initial state

**Theorem 16.11** (Continuous Polling)**.**
*Let $E = \textit{emailSystem}(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model with $a$ corresponding to the IOLTS given in Figure 16.1 and $u$ corresponding to the IOLTS given in Figure 16.8. Then the following holds:*

$$(\boldsymbol{AG}((\textit{client.state} = \text{unsigned}) \rightarrow (\textit{user.action} = \text{poll})))$$

If mail has arrived, the user will not poll for new mail until the email client is in mail read mode.

**Theorem 16.12** (Stop Polling)**.**
*Let $E = \textit{emailSystem}(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model with*

*$a$ corresponding to the IOLTS given in Figure 16.1 and $u$ corresponding to the IOLTS given in Figure 16.8. Then the following holds:*

$$(\mathbf{AG}((\mathit{client.state} = \text{notChecked}) \rightarrow \\ \mathbf{AX}(\mathbf{A}[(\neg(\mathit{user.action} = \text{poll}))\mathbf{U}(\mathit{client.state} = \text{mailNotChanged})])))$$

Once the user is reading mail, he will be moving the cursor forever.

**Theorem 16.13** (Moving Cursor When Reading)**.**

$$(\mathbf{AG}((\mathit{client.state} = \text{mailNotChanged}) \rightarrow \\ \mathbf{AG}(\mathit{user.action} = \text{moveCursor})))$$

If new mail has arrived and the signature of the current mail has not been checked, the user will not poll for new mail until the signature has been checked

If the user is in read email mode, he either was already reading email in the step before, or the signature had been checked.

**Theorem 16.14** (Signature Checked Before Reading)**.**
*Let $E = \mathit{emailSystem}(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model with $a$ corresponding to the IOLTS given in Figure 16.1 and $u$ corresponding to the IOLTS given in Figure 16.8. Then the following holds:*

$$\mathbf{AG}((\mathbf{EX}\mathit{client.state} = \text{mailNotChanged}) \rightarrow \\ ((\mathit{client.state} = \text{mailNotChanged}) \vee (\mathit{client.state} = \text{checked})))$$

**Proof**

Appendix F gives the conjunction of theorems 16.9 to 16.14 in the input format of theNuSMV (Cimatti et al., 2002) model checker. Correctness has been proven in NuSMV 2.3.0. □

### 16.2.2 Hoare Part

We want so show that the user can read an arbitrary email. For this, we show that it is always possible for the user to move the cursor over the whole email message. The idea behind this is that if the user will have seen each character of the message in the correct order, he has understood the email message. In Section 16.1.1 we have already shown that it is possible to reach the first position of the email by repeatedly issuing command `cursorLeft`. Here, we show that the user will have traversed all characters of the email message in the right order if he repeatedly issues command `cursorRight` starting from cursor position $(0,0)$. A specification of procedure `cursorRight` is given in Figure 16.9.

**context** `cursorRight()`
**post** **if** *flatCPos*(`m.st`,`m.cx`,`m.cy`) $<$ *len*(*flatten*(`m.st`))
**then**
*flatCPos*(`m.st`,`m.cx`,`m.cy`) =
*flatCPos*(`m.st@pre`,`m.cx@pre`,`m.cy@pre`) + 1
result = RESULT_CURSOR_MOVED
**else**
result = RESULT_MOVE_CURSOR_FAILED
**end if**

Figure 16.9: Specification of `cursorRight`

**Theorem 16.15.**
*Let $E =$ emailSystem$(U, A, I, O, M, u, a, \lambda, k, t)$ be an email system model with $a$ corresponding to the IOLTS given in Figure 16.1 and `insertChar` as specified in Figure 16.6. Let `m.st` be the list of strings representing the email with `m.st` $= []$, `m.cx` the $x$ position of the cursor and `m.cy` the $y$ position with `m.cx` $= 0$ and `m.cy` $= 0$ Let message be the message the user wants to enter.*

*If the user executes `cursorRight` sufficiently often, then the cursor will move over all elements of flatten(`m.st`) consecutively.*

**Proof**

We prove the theorem by induction over the length of email message `m.st` with the induction hypothesis that the user has read the email up to the $n$th character

**Base case: n = 0** This case is trivally true.

**Induction step: The user has read the email up to position $n$, and *flatCPos* is $n$** From the definition of `cursorRight` it follows that the cursor is at position $n+1$ in the next step. Since the user has read the first $n$ characters in order and reads the $(n+1)$th character next, he has read $n+1$ characters in order. □

# Chapter 17

# Conclusions

We have successfully applied the formal methodology developed in Parts I and II to the specification and verification of a secure email client. In the project Verisoft (`http://www.verisoft.de`), an actual email client satisfying the specification has been developed, and correctness has been proven. With this example application, we have shown how formal methods can be used to guarantee a fundamental requirement of user interface security. Our approach provides both formal definitions of HCI security and a formal method for the pervasive specification and verification of interactive applications. Developing a methodology for the pervasive specification of secure interactive applications posed a number of challenges:

- A formal methodology for the description of human-computer interaction had to be developed.
- Security requirements for HCI had to be developed and formalized.
- Theorems proven in different formal methods had to be integrated.

In Part I, the formal methodology for the description of HCI has been developed. Our methodology is based on IOLTS as a formal modeling method, and GOMS as a method for the description of user interfaces. We have developed a formal semantic for GOMS. By introducing hierarchical models, it becomes possible to describe HCI on any level of granularity. In order to allow for the pervasive formal description of human-computer interaction, the IOLTS based methodology has been integrated with Hoare logics procedure descriptions.

In Part II, formal security criteria for human-computer interaction have been developed. The formal criteria are based on the Common Criteria, a standardized international computer security requirement catalog, and the established definition of computer security as Confidentiality, Integrity, and Availability (CIA). We adapted both the Common Criteria and CIA to user interface security, formalized

the relevant concepts, and evaluated the relationships between concepts from the Common Criteria and CIA concepts.

In Part III, the methodology has been applied to the specification and verification of a secure email client. We have shown how Confidentiality, Integrity, and Availability are guaranteed for the email client. In the Verisoft project, an implementation of the email client has been developed and the procedural correctness has been proven. The results from Part III are not only relevant for the Verisoft email client. The design pattern in Chapters 14 and Chapter 15 are generic. If a specification follows the design pattern, then integrity and confidentiality are guaranteed.

# Part IV

# Appendices

# Appendix A

# First eVoting Example

The SMV file in Section A.1 implements the basic version of the eVoting application and user model as given in Chapter 4. The trace given in Section A.2 shows that in the given model, it is possible that a final state is never reached, i.e. the basic model is faulty. In the trace, the user continously selects a candidate and cancels the selection in the next step. A final state is never reached.

## A.1 SMV File

```
MODULE voterComponent
VAR
  state: { start, chosen, confirmed, error} ;
  action: { chooseBob, cancelVote, confirmVote, idle } ;
ASSIGN
  init(state) := start;
  next(state) :=
    case
      state = start     : chosen ;
      state = chosen  & (action = confirmVote)  : confirmed ;
      state = chosen  & (action = cancelVote)  : start ;
      state = confirmed : confirmed ;
      1 : error ;
    esac;
  init(action) := chooseBob ;
  next(action) :=
    case
      state = start    : { confirmVote, cancelVote } ;
      state = chosen & (action = confirmVote) : idle ;
      state = chosen & (action = cancelVote) : chooseBob ;
```

```
      1 : idle ;
    esac;

MODULE votingComputerComponent(userInput)
VAR
  state: { unlocked, voteCastBob, voteConfirmedBob, error} ;
ASSIGN
  init(state) := unlocked;
  next(state) :=
    case
      (state = unlocked)    & (userInput = chooseBob)
        : voteCastBob ;
      (state = voteCastBob) & (userInput = cancelVote)
        : unlocked ;
      (state = voteCastBob) & (userInput = confirmVote)
        : voteConfirmedBob ;
      state = voteConfirmedBob : voteConfirmedBob ;
      1 : error ;
    esac;
DEFINE
  final := state = voteConfirmedBob ;

MODULE main
VAR
  voter :  voterComponent();
  votingComputer : votingComputerComponent(voter.action);

SPEC
  AG (votingComputer.state != error)
  & AG (voter.state != error)
  & AF (votingComputer.final)
```

## A.2 Refutation Generated by NuSMV

```
-- specification  ((AG votingComputer.state != error &
                     AG voter.state != error)
                    & AF votingComputer.final)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
```

```
  voter.state = start
  voter.action = chooseBob
  votingComputer.state = unlocked
  votingComputer.final = 0
-> State: 1.2 <-
  voter.state = chosen
  voter.action = cancelVote
  votingComputer.state = voteCastBob
-> State: 1.3 <-
  voter.state = start
  voter.action = chooseBob
  votingComputer.state = unlocked
```

## A.3 Message Trace Example

This example from Chapter 9 shows how temporal properties about messages are formalized. The specification guarantees that the action "cancelVote" is never immediately followed by the action "confirmVote." The NuSMV code is identical to the code in Appendix A.1, except for the SPEC declaration:

```
SPEC
  AG (votingComputer.state != error)
  & AG (voter.state != error)
  & AG (voter.action = cancelVote ->
          AX voter.action != confirmVote)
```

# Appendix B

# Perl Program converting GOMS to IOLTS

This Perl program implements Algorithms 1 and 2 from Chapter 5. It takes formal GOMS models (Definition 5.1 in Chapter 5) and generates a corresponding IOLTS.

```
#!/usr/bin/perl -w

use strict;

# Define GOMS model

my @G_c = ("VOTE_FOR_CANDIDATE_c", "CHANGE_VOTE",
           "CHANGE_VOTE_2", "REVIEW_VOTE_2", "REVIEW_VOTE_3");
my @O_c = ("WAIT_FOR_UNLOCK", "CHOOSE_CANDIDATE_c",
           "CONFIRM_VOTE", "CANCEL_VOTE", "FAIL");
my @m_c_0 = ("VOTE_FOR_CANDIDATE_c", "WAIT_FOR_UNLOCK",
             "CHOOSE_CANDIDATE_c", "REVIEW_VOTE");
my @m_c_1 = ("CHANGE_VOTE", "CANCEL_VOTE",
             "CHOOSE_CANDIDATE_c", "REVIEW_VOTE_2");
my @m_c_2 = ("CHANGE_VOTE_2", "CANCEL_VOTE",
             " "CHOOSE_CANDIDATE_c", REVIEW_VOTE_3");
my @M_c = (\@m_c_0, \@m_c_1, \@m_c_2);
my @C_c = ("c_selected", "not_c_selected");
my @r_c_0 = ("REVIEW_VOTE", "c_selected", "CONFIRM_VOTE");
my @r_c_1 = ("REVIEW_VOTE", "neg_c_selected", "CHANGE_VOTE");
my @r_c_2 = ("REVIEW_VOTE_2", "c_selected", "CONFIRM_VOTE");
my @r_c_3 = ("REVIEW_VOTE_2", "neg_c_selected",
```

```
             "CHANGE_VOTE_2");
my @r_c_4 = ("REVIEW_VOTE_3", "c_selected", "CONFIRM_VOTE");
my @r_c_5 = ("REVIEW_VOTE_3", "neg_c_selected", "FAIL");
my @R_c = (\@r_c_0, \@r_c_1, \@r_c_2, \@r_c_3, \@r_c_4,
           \@r_c_5);
my $g_0_c = "VOTE_FOR_CANDIDATE_c";

# Start recursion

my $state_counter = 0;
my @s_0 = (&new_state("s",0));
my ($S, $Sigma_in, $Sigma_out, $transition, $s_last) =
  &sub_goms_to_iolts (\@G_c, \@O_c, \@M_c, \@R_c, \@C_c,
                      $g_0_c, \@s_0, "s");
for(my $i = 0; defined ($transition->[$i]); $i++){
  print "(" . $transition->[$i]->[0] . ", "
    . $transition->[$i]->[1] . ", "
      . $transition->[$i]->[2] . ")\n";
}

#########################################################
#
# sub sub_goms_to_iolts($$$$$)
#
#########################################################

sub sub_goms_to_iolts($$$$$$){

  my ($G, $O, $M, $R, $C, $g_0, $s_0,
      $state_name_prefix) = @_;
  my (@S, @Sigma_in, @Sigma_out, @transition, @s_last);

  # \IF{$g_0 \in O$}
  if(&is_element($g_0, $O)){
    # Operator
    my $s_next = new_state($state_name_prefix,0);
    for(my $i = 0; (defined $s_0->[$i]); $i++){
      push @transition, [$s_0->[$i], $g_0, $s_next];
    }
    @s_last = ( $s_next );
  }
```

```
else{
  # Goal
  my $m = &find_matching_method($g_0,$M);
  if(defined $m){
    my $s_last_i;
    for(my $i = 1; (defined $m->[$i]); $i++){
      my ($S_i, $Sigma_in_i, $Sigma_out_i, $transition_h_i);
      ($S_i, $Sigma_in_i, $Sigma_out_i, $transition_h_i,
       $s_last_i) =
        &sub_goms_to_iolts($G, $O, $M, $R, $C, $m->[$i],
                           $s_0, "${state_name_prefix}_${i}");
      for(my $l = 0; defined($transition_h_i->[$l]); $l++){
        push @transition, $transition_h_i->[$l];
      }
      $s_0 = $s_last_i;
    }
    for(my $i = 0; (defined $s_last_i->[$i]); $i++){
      push @s_last, $s_last_i->[$i];
    }
  }
  else{
    # Selection
    @s_last = ();
    for(my $i = 0; (defined $R->[$i]); $i++){
      my $g = $R->[$i]->[0];
      my $c = $R->[$i]->[1];
      my $g_prim = $R->[$i]->[2];
      if($g eq $g_0){
        my @s_next = (new_state($state_name_prefix,$i));
        for(my $i = 0; (defined $s_0->[$i]); $i++){
          push @transition, [$s_0->[$i], $c, $s_next[0]];
        }
        my ($S_i, $Sigma_in_i, $Sigma_out_i,
            $transition_h_i, $s_last_i) =
              &sub_goms_to_iolts($G, $O, $M, $R, $C, $g_prim,
                                 \@s_next,
                                "${state_name_prefix}_${i}");
        for(my $l = 0; defined($transition_h_i->[$l]); $l++){
          push @transition, $transition_h_i->[$l];
        }
        for(my $i = 0; (defined $s_last_i->[$i]); $i++){
```

```
            push @s_last, $s_last_i->[$i];
          }
        }
      }
    }
  }
  return (\@S, \@Sigma_in, \@Sigma_out, \@transition,
          \@s_last );
}

sub is_element($$){
  my ($e, $a) = @_;
  my $i;
  for($i = 0; (defined($a->[$i]) && (($a->[$i]) ne $e));
      $i++){ };
  return (defined($a->[$i]));
}

############################################################
#
# sub new_state($$)
#
############################################################

sub new_state(){
  my ($prefix, $num) = @_;
  return ($prefix . "_" . $num);
}


############################################################
#
# sub find_matching_method($$)
#
############################################################

sub find_matching_method($$){
  my ($g, $M) = @_;
  my $i;
  for($i = 0;
      (defined($M->[$i]) && (($M->[$i]->[0]) ne $g));
```

```
      $i++){ };
  return ($M->[$i]);
}
```

# Appendix C

# Basic Main Execution Cycle Model

This chapter provides NuSMV code of the models using the naïve execution loops developed in Chapter 14. All system models in Chapter 14 use the same user and application logic components, and the same screen component. These are given in Section C.1 and Section C.3, respectively. Section C.2 contains the naïve user and application execution loops. In Section C.4, the components are combined to three models. In the first model (Section C.4.1), the user and application logic components are connected directly to each other. For this system, the integrity constraint is satisfied. In the second model (Section C.4.2), the user and application logic components are connected via user and application execution loop components. The output of the application execution component connects directly to the input of the user execution component; there is no screen in between. For this system, the integrity constraint is satisfied. Finally, the second model is extended by a screen component in Section C.4.3. The asynchroneity introduced by the screen component leads to a model which does not satisfy the integrity constraint. A refutation trace generated by NuSMV is given in Section C.5.

## C.1 Logic Components

### C.1.1 Application Logic Component

```
MODULE applicationLogicComponent(CMD)
VAR
  state: { NoNewMail, NewMail} ;
  AppOut: { ScreenNoMail, ScreenNewMail, idle } ;
ASSIGN
  init(state) := NoNewMail;
  next(state) :=
```

```
    case
      (CMD = idle) : state;
      1 : {NoNewMail, NewMail} ;
    esac;
  init(AppOut) :=  ScreenNoMail;
  next(AppOut) :=
    case
      (next(state) = NoNewMail) :  ScreenNoMail ;
      (next(state) = NewMail) :  ScreenNewMail ;
    esac ;
```

### C.1.2 User Logic Component

```
MODULE userLogicComponent(appOut)
VAR
  state: { NoNewMail, NewMail} ;
  CMD: { CMD_Check } ;
ASSIGN
  init(state) := NoNewMail;
  next(state) :=
    case
      (next(appOut) = ScreenNoMail)  :  NoNewMail ;
      (next(appOut) = ScreenNewMail) :  NewMail ;
      1 : state ;
    esac ;
```

## C.2 Execution Loop Components

### C.2.1 Basic Application Execution Loop Component

```
MODULE simpleAppExecComponent(CMDIn, AppOutIn)
VAR
  state: { CmdExecuted, GotResult, ResultShown, GotCmd } ;
  AppOutOut: { ScreenNoMail, ScreenNewMail, idle, Ready,
               idle } ;
  CMDOut: { CMD_Check, idle } ;
ASSIGN
  init(state) := GotResult;
  next(state) :=
    case
```

```
      (state = CmdExecuted) & (next(AppOutIn) != idle)
        :  GotResult ;
      (state = GotResult) :  ResultShown ;
      (state = ResultShown) & (next(CMDIn) != idle)
        : GotCmd ;
      (state = GotCmd) : CmdExecuted ;
      1 : state;
    esac ;
  init(CMDOut) := idle ;
  next (CMDOut) :=
    case
      (state = GotCmd) : CMDIn ;
      1 : idle;
    esac;
  init(AppOutOut) := idle;
  next (AppOutOut) :=
    case
      (state = GotResult) : AppOutIn ;
      1 : idle;
    esac;
```

### C.2.2 Basic User Execution Loop Component

```
MODULE simpleUserExecComponent(CMDIn, AppOutIn)
VAR
  state: { Decided, Wait, StartClassify, EndClassify } ;
  AppOutOut: { ScreenNoMail, ScreenNewMail, Processing,
               Ready, idle } ;
  CMDOut: { CMD_Check, idle } ;
ASSIGN
  init(state) := Wait;
  next(state) :=
    case
      (state = Wait) & (next(AppOutIn) != idle)
        :  StartClassify ;
      (state = StartClassify) :  EndClassify ;
      (state = EndClassify) & (next(CMDIn) != idle)
        : Decided ;
      (state = Decided) : Wait ;
      1 : state;
    esac ;
```

```
  init(CMDOut) := idle ;
  next (CMDOut) :=
    case
      (state = Decided) : CMDIn ;
      1 : idle;
    esac;
  init(AppOutOut) := idle;
  next (AppOutOut) :=
    case
      (state = StartClassify) : AppOutIn ;
      1 : idle;
    esac;
```

## C.3 Other Components

### C.3.1 Screen

```
MODULE screenComponent(ScreenIn)
VAR
  ScreenOut: { ScreenNoMail, ScreenNewMail, Processing,
               Ready, idle } ;
ASSIGN
  init(ScreenOut) := ScreenNoMail;
  next(ScreenOut) :=
    case
      (ScreenIn = idle) : ScreenOut;
      1                 : ScreenIn;
    esac;
```

## C.4 Models

### C.4.1 Direct Connection of Logic Components

```
-- The model connects the user logic component and the
-- application logic component directly to each other.

#include "../logics/app_logic.smv"
#include "../logics/user_logic.smv"

MODULE main
```

```
VAR
  appLogic :  applicationLogicComponent(userLogic.CMD);
  userLogic : userLogicComponent(appLogic.AppOut);

SPEC
  AG (appLogic.state = userLogic.state)
```

### C.4.2 Basic System Model without Screen

```
-- This model implements the basic execution loop with
-- the user and application logic components. The
-- integrity constraint is satisfied, because screen
-- output is directly relayed to the user, without a
-- screen in between.

#include "../logics/app_logic.smv"
#include "../logics/user_logic.smv"
#include "../execs/simple_app_exec.smv"
#include "../execs/simple_user_exec.smv"

MODULE main
VAR
  appLogic  : applicationLogicComponent(appExec.CMDOut);
  appExec   : simpleAppExecComponent(userExec.CMDOut,
                                     appLogic.AppOut);
  userLogic : userLogicComponent(userExec.AppOutOut);
  userExec  : simpleUserExecComponent(userLogic.CMD,
                                      appExec.AppOutOut);

SPEC
  AG ((userExec.state = EndClassify) ->
        (appLogic.state = userLogic.state))
```

### C.4.3 Basic System Model with Screen

```
-- This model implements the basic execution loop with
-- the user and application logic components. A screen
-- is in between the output of the application and the
-- other. The integrity constraint is not satisfied.
```

```
#include "../logics/app_logic.smv"
#include "../logics/user_logic.smv"
#include "../execs/simple_app_exec.smv"
#include "../execs/simple_user_exec.smv"
#include "../other/screen.smv"

MODULE main
VAR
  appLogic  : applicationLogicComponent(appExec.CMDOut);
  userLogic : userLogicComponent(userExec.AppOutOut);
  appExec   : simpleAppExecComponent(userExec.CMDOut,
                                     appLogic.AppOut);
  userExec  : simpleUserExecComponent(userLogic.CMD,
                                      screen.ScreenOut);
  screen    : screenComponent(appExec.AppOutOut);

SPEC
  AG ((userExec.state = EndClassify) ->
        (appLogic.state = userLogic.state))
```

## C.5 Refutation of System Model with Screen

```
-- specification  AG (userExec.state = EndClassify ->
--   appLogic.state = userLogic.state)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  appLogic.state = NoNewMail
  appLogic.AppOut = ScreenNoMail
  userLogic.state = NoNewMail
  appExec.state = GotResult
  appExec.AppOutOut = idle
  appExec.CMDOut = idle
  userExec.state = Wait
  userExec.AppOutOut = idle
  userExec.CMDOut = idle
  screen.ScreenOut = ScreenNoMail
  userLogic.CMD = CMD_Check
-> State: 1.2 <-
```

```
  appExec.state = ResultShown
  appExec.AppOutOut = ScreenNoMail
  userExec.state = StartClassify
-> State: 1.3 <-
  appExec.AppOutOut = idle
  userExec.state = EndClassify
  userExec.AppOutOut = ScreenNoMail
-> State: 1.4 <-
  userExec.state = Decided
  userExec.AppOutOut = idle
-> State: 1.5 <-
  appExec.state = GotCmd
  userExec.state = Wait
  userExec.CMDOut = CMD_Check
-> State: 1.6 <-
  appExec.state = CmdExecuted
  appExec.CMDOut = CMD_Check
  userExec.state = StartClassify
  userExec.CMDOut = idle
-> State: 1.7 <-
  appLogic.state = NewMail
  appLogic.AppOut = ScreenNewMail
  appExec.state = GotResult
  appExec.CMDOut = idle
  userExec.state = EndClassify
  userExec.AppOutOut = ScreenNoMail
```

# Appendix D

# Improved Main Execution Cycle Model

In Chapter 14.2, we have shown that the naïve user and application execution components may lead to violations of the integrity constraint. NuSMV code generating a refutation has been given in Appendix C. Here, we provide NuSMV code implementing the improved user and application execution components developed in Chapter 14.2. With these components, the integrity constraint is satisfied.

## D.1 Logic Components

The user logic component and the application logic component are identical to the ones from Appendix C.1.

## D.2 Execution Loop Components

### D.2.1 Improved Application Execution Loop Component

```
MODULE appExecComponent(CMDIn, AppOutIn)
VAR
  state: { CmdExecuted, GotResult, ResultShown, GotCmd,
           Busy, Escaped } ;
  AppOutOut: { ScreenNoMail, ScreenNewMail, Processing,
               Ready, idle } ;
  CMDOut: { CMD_Check, ESC, idle } ;
ASSIGN
  init(state) := GotResult;
  next(state) :=
```

```
    case
      (state = CmdExecuted) & (next(AppOutIn) != idle)
        :  GotResult ;
      (state = GotResult) :  ResultShown ;
      (state = ResultShown) & (!(next(CMDIn) in {idle, ESC}))
        : Busy ;
      (state = ResultShown) & (next(CMDIn) = ESC)
        : Escaped ;
      (state = Escaped) : ResultShown ;
      (state = Busy) : GotCmd ;
      (state = GotCmd) : CmdExecuted ;
      1 : state;
    esac ;
  init(CMDOut) := idle ;
  next (CMDOut) :=
    case
      (state = Busy) : CMDIn ;
      1 : idle;
    esac;
  init(AppOutOut) := idle;
  next (AppOutOut) :=
    case
      (state = GotResult) : AppOutIn ;
      (state = Busy) : Processing ;
      (state = Escaped) : Ready ;
      1 : idle;
    esac;
```

### D.2.2 Improved User Execution Loop Component

```
MODULE userExecComponent(CMDIn, AppOutIn)
VAR
  state: { Decided, BeginWait, EndWait, StartClassify,
           EndClassify, Escape } ;
  AppOutOut: { ScreenNoMail, ScreenNewMail, Processing,
               Ready, idle } ;
  CMDOut: { CMD_Check, ESC, idle } ;
ASSIGN
  init(state) := EndWait;
  next(state) :=
    case
```

```
      (state = EndWait)
        & (!(next(AppOutIn) in {idle, Processing}))
        : StartClassify ;
      (state = StartClassify) :  EndClassify ;
      (state = EndClassify) & (next(CMDIn) != idle)
        : Decided ;
      (state = Decided) : BeginWait ;
      (state = BeginWait) & (next(AppOutIn) = Processing)
        : EndWait ;
      (state = BeginWait) : Escape ;
      (state = Escape) & (next(AppOutIn) = Ready)
        : StartClassify ;
      1 : state;
    esac ;
  init(CMDOut) := idle ;
  next (CMDOut) :=
    case
      (state = Decided) : CMDIn ;
      (state = BeginWait) & (next(state) = Escape) : ESC ;
      1 : idle;
    esac;
  init(AppOutOut) := idle;
  next (AppOutOut) :=
    case
      (state = StartClassify) : AppOutIn ;
      1 : idle;
    esac;
```

# D.3 Other Components

## D.3.1 Screen

The screen component is identical to the one from Appendix C.3.1.

# D.4 Model

## D.4.1 Improved System Model with Screen

```
-- In the improved model, the integrity constraint is
--  satisfied for the model including a screen, too.
```

```
#include "../logics/app_logic.smv"
#include "../logics/user_logic.smv"
#include "../execs/user_exec.smv"
#include "../execs/app_exec.smv"
#include "../other/screen.smv"

MODULE main
VAR
  appLogic  : applicationLogicComponent(appExec.CMDOut);
  userLogic : userLogicComponent(userExec.AppOutOut);
  appExec   : appExecComponent(userExec.CMDOut,
                               appLogic.AppOut);
  userExec  : userExecComponent(userLogic.CMD,
                                screen.ScreenOut);
  screen    : screenComponent(appExec.AppOutOut);

SPEC
  AG ((userExec.state = EndClassify) ->
        (appLogic.state = userLogic.state))
```

# Appendix E

# Writing, Signing, Sending Email

In Chapter 16, two availability requirements for the email client have been defined. The user should be able to write, sign, and send email, and the user should be able to poll for new email, check the signature of received email, and read received email. Here, we provide NuSMV code to show that the first availability requirement is satisfied. The model makes use of the application logic component and the screen component defined in Appendix C, and the user and application execution components defined in Appendix D. The model presented here introduces a model of a user writing, signing, and sending email.

```
-- User model to show that it is possible to write,
-- sign and send email.

MODULE emailUserComponent
VAR
  state: {initialize, editing, signing, sending } ;
  action: { sign, send, insertPubKey, generateKeyPair,
            editMail, moveCursor, insertChar, deleteChar,
            quitEditing, checkSig, poll, editPubKey, idle } ;
ASSIGN
  init(state) := initialize ;
  next(state) :=
    case
      (state = initialize) : editing ;
      (state = editing) : {editing, signing } ;
      (state = signing) : sending ;
      (state = sending) : initialize ;
    esac;
  init(action) := editMail ;
```

```
  next(action) :=
    case
      (next(state) = editing) : { moveCursor, insertChar,
                                  deleteChar} ;
      (next(state) = signing ) : { quitEditing } ;
      (next(state) = sending ) : { sign } ;
      (next(state) = initialize)  : { send } ;
    esac;


#include "email_client.smv"

MODULE main
VAR
  user :  emailUserComponent();
  client : emailClientComponent(user.action);

SPEC
 -- From the initial state, the cursor can be
 -- moved an arbitrary number of times while being in
 -- state mailNotChanged,
 EF (EG ((user.action = moveCursor)
         & (client.state = mailNotChanged)))
 -- and if the client is in state mailNotChanged,
 --  it is possible to transit to state mailChanged in
 --  the next step,
 & EF ((client.state = mailNotChanged)
       & EX (client.state = mailChanged))
 -- and if the client is in state mailChanged, an
 -- arbitrary number of characters can be deleted,
 & EF (EG ((user.action = deleteChar)
           & (client.state = mailChanged)))
 -- and if the client is in state mailChanged, an
 -- arbitrary number of characters can be inserted,
 & EF (EG ((user.action = insertChar)
           & (client.state = mailChanged)))
 -- and if the client is in state mailChanged, state sent
 -- is reachable without transiting through mailChanged
 -- again,
 & EF ((client.state = mailChanged) &
       EX (E [(client.state != mailChanged)
```

```
          U (client.state = sent)]))
-- and if sent is reached, then the previous state must
-- be signed.
& AG ((EX (client.state = sent )) ->
        (client.state = signed))
```

# Appendix F

# Receiving, Checking, Reading Email

In Chapter 16, two availability requirements for the email client have been defined. The user should be able to write, sign, and send email, and the user should be able to poll for new email, check the signature of received email, and read received email. Here, we provide NuSMV code to show that the second availability requirement is satisfied. The model makes use of the application logic component and the screen component defined in Appendix C, and the user and application execution components defined in Appendix D. The model presented here introduces a model of a user polling for new email, checking the signature, and reading the email.

```
-- User model to show that it is possible to receive email,
-- check signatures, and read email.

MODULE emailUserComponent(clientOutput)
VAR
  state: {Polling, got_new_mail, signature_checked,
          readingMail } ;
  action: { sign, send, insertPubKey, generateKeyPair,
            editMail, moveCursor, insertChar, deleteChar,
            quitEditing, checkSig, poll, editPubKey, idle } ;
ASSIGN
  init(state) := Polling ;
  next(state) :=
    case
      (state = Polling) & (clientOutput = notChecked)
        : got_new_mail ;
      (state = Polling) & (clientOutput != notChecked)
        : Polling ;
```

```
      (state = got_new_mail) : signature_checked ;
      (state = signature_checked) : readingMail ;
      (state = readingMail) : readingMail ;
    esac;
  init(action) := poll ;
  next(action) :=
    case
      (state = Polling) & (clientOutput = notChecked)
        : checkSig ;
      (state = Polling) & (clientOutput != notChecked)
        : poll ;
      (state = got_new_mail) : editMail ;
      (state = signature_checked) : moveCursor ;
      (state = readingMail) : moveCursor ;
    esac;


#include "email_client.smv"

MODULE main
VAR
  user :  emailUserComponent(client.state);
  client : emailClientComponent(user.action);

SPEC
 -- If the system is in the initial state and new
 -- mail is available, then both the new mail will have
 -- arrived in the next step and the user will know about.
 (AG (((client.state = unsigned) & client.newMailAvailable)
         -> AX ((client.state = notChecked)
                & AX (user.state = got_new_mail))))
 -- If the system is in the initial state and no mail is
 -- available, then both the system and the user will
 -- remain in their initial states in the next step.
 & (AG (((client.state = unsigned) & !client.newMailAvailable)
         -> AX ((client.state = unsigned)
                & AX (user.state = Polling))))
 -- The user will continue polling for new mail while the
 -- system is in its initial state
 & (AG ((client.state = unsigned) -> (user.action = poll)))
 -- If mail has arrived, the user will not poll for new mail
```

```
--   until the email client is in mail read mode.
& (AG ((client.state = notChecked) ->
         AX (A [ (!(user.action = poll))
             U (client.state = mailNotChanged)])))
-- Once the user is reading mail, he will be moving the
--  cursor forever.
& (AG ((client.state = mailNotChanged) ->
       AG (user.action = moveCursor)))
-- If the user is in read email mode, he either
-- was already reading email in the step before, or the signature
--  had been checked.
& AG ((EX client.state = mailNotChanged) ->
      ((client.state = mailNotChanged) | (client.state = checked)))
```

# Bibliography

G. D. Abowd and R. Beale. Users, systems and interfaces: A unifying framework for interaction. In D. Diaper and N. Hammond, editors, *HCI'91:People and Computers VI*, pages 73–87. Cambridge University Press, 1991.

G. D. Abowd, J. P. Bowen, A. J. Dix, M. D. Harrison, and R. Took. User interface languages: A survey of existing methods. Technical Report PRG-TR-5-89, Oxford University Computing Laboratory, October 1989.

Nathaniel Ayewah, Sven Beyer, Nikhil Kikkeri, and Peter-Michael Seidel. Challenges in the formal verification of complete state-of-the-art processors. In *International Conference on Computer Design*, San Jose, 2005.

Daniel Bachfeld. Nepper, Schlepper, Bauernfänger — Risiken beim Online-Banking. *c't magazin für Computertechnik*, 22:148–153, 2005.

Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc. ISBN 3-540-42124-6.

Bernhard Beckert and Gerd Beuster. Formal specification of security-relevant properties of user interfaces. In *Proceedings, 3rd International Workshop on Critical Systems Development with UML, Lisbon, Portugal*, Munich, Germany, 2004. TU Munich Technical Report TUM-I0415.

Bernhard Beckert and Gerd Beuster. Guaranteeing consistency in text-based human-computer interaction. In *Pre-event Proceedings of the 1st International Workshop on Formal Methods for Interactive Systems (FMIS 2006), Macao SAR China*. The United Nations University, 2006a. UNU-IIST Report No. 347.

Bernhard Beckert and Gerd Beuster. Guaranteeing consistency in text-based human-computer interaction. *Special Issue of Innovations in System and Software Engineering (Submitted)*, 2007.

Bernhard Beckert and Gerd Beuster. A method for formalizing, analyzing, and verifying secure user interfaces. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, volume 4260 of *Lecture Notes in Computer Science*. Springer, 2006b. ISBN 3-540-47460-9.

Bernhard Beckert, Gerd Beuster, and Pia Breuer. TR #2: Email Client Specification. Technical report, Verisoft Konsortium, 2007.

Jean Berstel, Stefano Crespi Reghizzi, Gilles Roussel, and Pierluigi San Pietro. A scalable formal method for design and automatic checking of user interfaces. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14 (2):124–167, April 2005.

Gerd Beuster and Roman Neruda. Description and generation of computational agents. In *Proceedings of the First International Conference on Knowledge Science, Engineering and Management (KSEM'06)*. Springer, 2006.

Gerd Beuster, Pavel Krušina, Petra Kudová, Roman Neruda, and Pavel Rydvan. Towards building computational agent schemes. In *Proceedings of the International Conference on Genetic Algorithms and Artificial Neural Networks 2003 (ICANNGA 2003)*, Roanne, France, 2003.

Gerd Beuster, Pavel Krušina, Petra Kudová, Roman Neruda, and Pavel Rydvan. Bang 3: A computational multi-agent system. *IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'04)*, 00:563–564, 2004.

Gerd Beuster, Niklas Henrich, and Markus Wagner. Real world verification — experiences from the verisoft email client. In *Proceedings of the Workshop on Empirical Succesfully Computerized Reasoning (ESCoR 2006)*, 2006.

Simon P. Booth and Simon B. Jones. A screen editor written in the miranda functional programming language. Technical Report TR-116, Department of Computing Science and Mathematics, University of Stirling, February 1994.

S. Brackin. Evaluating and improving protocol analysis by automatic proof. In *CSFW '98: Proceedings of the 11th IEEE Computer Security Foundations Workshop*, page 138, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8488-7.

Thomas Browne, David Dávila, Spencer Rugaber, and Kurt Stirewalt. Using declarative descriptions to model user interfaces with mastermind. In F. Paterno and P. Palanque, editors, *Formal Methods in Human Computer Interaction*. Springer-Verlag, 1997.

P. Brun. Xtl: A temporal logic for the formal development of interactive systems. In Philippe Palanque, editor, *Formal methods in human computer interaction*. Springer, New York, London,. . . , 1998.

M. Burrows, M. Abadi, and R. Needham. A logic of authentication. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 1–13, New York, NY, USA, 1989. ACM Press. ISBN 0-89791-338-8.

M. Cabrera, M. Gea, F. Gutierrez, and J.C. Torres. Algebraic specification of user interfaces. In *1st ERCIM Workshop on "User Interfaces for All"*, Heraklion, Crete, Greece, 30–31 October 1995.

Gaelle Calvary, Joelle Coutaz, and Laurence Nigay. From single-user architectural design to PAC*: a generic software architecture model for CSCW. In *CHI*, pages 242–249, 1997.

David A. Carr. Interaction object graphs: an executable graphical notation for specifying user interfaces. In Philippe Palanque and Fabio Paternò, editors, *Formal methods in Human-Computer Interaction*, pages 141–155. Springer, 1997.

Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, June 2004.

A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. *sp*, 00:184, 1987. ISSN 1540-7993.

ITS. *Information Technology Security Evaluation Criteria (ITSEC) — Provisional Harmonised Criteria*. Commission of the European Communities, June 1991. ISBN 92-826-3004-8.

Common Criteria Evaluation Board (CCEB). *Common Criteria for Information Technology Security Evaluation (CC) — Version 3.1*, 2006.

Patrick Cousot and Radhia Cousot. Modular static program analysis. In N. Horspool, editor, *Proceedings of the International Conference on Compiler Construction (CC 2002)*, LNCS 2304, pages 159–178, Grenoble, France, April 6–14 2002.

W. Damm, H. Hungar, and E.-R. Olderog. On the verification of cooperating traffic agents. In F.S. de Boer, M.M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. FMCO '03: Formal Methods for Components and Objects*, LNCS 3188, pages 78–110, 2004.

Geert de Haan. *ETAG, A Formal Model of Competence Knowledge for User-Interface Design*. PhD thesis, Vrije Universiteit, Amsterdam, 2000.

Geert de Haan. Extended task-action grammar (ETAG): the psychological basis of a formal model for user interface design. http://home.tiscali.nl/gdehaan/etag.html, June 2005, 1995.

Rüdiger Dierstein. Sicherheit in der informationstechnik — der begriff it-sicherheit. *Informatik Spektrum*, 27(4), August 2004.

A. Dix and G. Abowd. Modelling status and event behaviour of interactive systems. *Software Engineering Journal*, 11(6):334–346, 1996.

A. J. Dix and C. Runciman. Abstract models of interactive systems. In P. Johnson and S. Cook, editors, *HCI'85: People and Computers I: Designing the Interface*, pages 13–22. Cambridge: Cambridge University Press, 1985.

Alan Dix, Janet Finley, Gregory Abowd, and Russell Beale, editors. *Human-computer interaction*, chapter 3. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998. http://www.hcibook.com/hcibook/downloads/pdf/slides.3.pdf.

DoD 5200.28-STD. *Department of Defense Trusted Computer System Evaluation Criteria*, December 1985.

Danny Dolev and Andrew C. Yao. On the security of public key protocols. Technical report, Stanford University, Stanford, CA, USA, 1981.

Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In *Proceedings, 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, LNCS 3603, pages 2–16. Springer, 2005.

P.C. Gilmore. A proof method for quantification theory. *IBM Journal of Research and Development*, 4:28–35, 1960.

Doug Goldson. Formal modelling of interactive systems. In *Proceedings of APAQS 2OOO, the First Asia-Pacific Conference on Quality Software*, IEEE Conference Proceedings. IEEE Computer Society Press, 2000.

Gerhard Goos and Wolf Zimmermann. Verification of compilers. In *Correct System Design*, pages 201–230, 1999.

Éric Goubault. Static analyses of floating-point operations. In P. Cousot, editor, *SAS'01*, LNCS 2126, pages 233–258, Paris, July 2001.

Jonathan Grudin. The case against user interface consistency. *Communications of the ACM*, 32(Issue 10):1164–1173, October 1989.

F. Hamilton. Predictive evaluation using task knowledge structures. In *Companion proceedings of CHI'96*, Vancouver, April 1996.

D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic Volume II — Extensions of Classical Logic*, pages 497–604. D. Reidel Publishing Company: Dordrecht, The Netherlands, 1984.

Michael Harrison, editor. *Formal methods in human-computer interaction*. Cambridge Univ. Press, Cambridge, Mass., 1990.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

A. Hussey and D. Carrington. Specifying a web browser interface using object-z. In Philippe Palanque, editor, *Formal methods in human computer interaction*, chapter 8. Springer, New York, London,. . . , 1998.

Vipul Jain. User interface description formalisms. Technical report, McGill University School of Computer Science, Montréal, Canada, 1994.

M. Jmaiel. Specifying communication protocols with temporal logic. Technical Report Technical Report 1994/16, Technische Universität Berlin, Fachbereich Informatik, 1994.

Bonnie E. John. Why GOMS? *interactions*, 2(4):80–89, October 1995. ACM Press, New York, NY, USA.

Bonnie E. John and David E. Kieras. The goms family of user interface analysis techniques: comparison and contrast. *ACM Trans. Comput.-Hum. Interact.*, 3 (4):320–351, 1996. ISSN 1073-0516.

Ron Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992. ISBN 0387562834.

Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1 (3):26–49, 1988. ISSN 0896-8438.

W. Kuhn and A.U. Frank. A formalization of metaphors and image-schemas in user interfaces. In D. M. Mark and A. U. Frank, editors, *Cognitive and Linguistic Aspects of Geographic Space*, NATO ASI Series. Kluwer Academic Press, Dordrecht, The Netherlands, 1991.

Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a c0 compiler. In *Proceedings, 3rd International Conference on Software Engineering and Formal Methods (SEFM 2005)*, Koblenz, Germany, 2005.

Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055, pages 147–166. Springer-Verlag, Berlin Germany, 1996.

Catherine Meadows. Formal methods for cryptographic protocol analysis: emerging issues and trends. *IEEE Journal on Selected Areas in Communications*, 21 (1):44–54, January 2003.

Allen Newell. *Unified theories of cognition*. Harvard University Press, Cambridge, MA, USA, 1994. ISBN 0-674-92101-1.

Object Modeling Group. *Unified Modelling Language Specification, version 1.5*. Object Modeling Group, March 2003. URL `http://www.omg.org/cgi-bin/doc?formal/03-03-01`.

Judith Reitman Olson and Gary M. Olson. *Human-computer interaction: toward the year 2000*, chapter The growth of cognitive modeling in human-computer interaction since GOMS, pages 603–625. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995. ISBN 1-55860-246-1.

P. Palanque and R. Bastide. Petri net based design of user-driven interfaces using the cooperative object formalism. In F. Paternó, editor, *Design, Specification and Verification of Interactive Systems '94*, pages 383–400, Heidelberg, 1994. Springer-Verlag.

Philippe Palanque and Fabio Paternò, editors. *Formal methods in human computer interaction*. Springer, New York, London,. . . , 1998.

Philippe Palanque, Remi Bastide, and Valerie Senges. Validating interactive system design through the verification of formal task and system models. In *Engineering for Human-Computer Interaction*. Chapman & Hall, August 1995.

Wolfgang Paul. Towards a worldwide verification technology. In *Proceedings of the Verified Software: Theories, Tools, Experiments Conference (VSTTE 2005)*, Zurich, Switzerland, October 2005.

Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

Robert W. Reeder and Roy A. Maxion. User interface dependability through goal-error prevention. In *DSN '05: Proceedings of the 2005 International Conference on Depen dable Systems and Networks (DSN'05)*, pages 60–69, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2282-3.

John Rushby. Using model checking to help discover mode confusions and other automation surprises. *Reliability Engineering and System Safety*, 75(2):167–177, February 2002.

Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452 of *LNAI*, pages 398–414. Springer, 2005.

E. Schlungbaum and T. Elwert. Modellierung von graphischen Benutzungsoberflächen im Rahmen des TADEUS-Ansatzes. In H.-D. Böcker, editor, *Software-Ergonomie'95: Mensch-Computer-Interaktion Anwendungsbereiche lernen voneinander*, pages 331–348. Teubner, Stuttgart, 1995.

S. Smith and D. Duke. Using csp to specify interaction in virtual environments. Technical report, University of York, 1999.

SSCD-PP. *Protection Profile — Secure Signature-Creation Device Type 3, Version 1.05*, 2001. http://www.bsi.bund.de/cc/pplist/PP0006b.pdf.

Artem Starostin. Formal verification of a c-library for strings. Master's thesis, Saarland University, 2006.

Bettina Sucrow. Formal specification of human-computer interaction by graph grammar s under consideration of information resources. In *Automated Software Engineering*, pages 28–35, 1997. URL `citeseer.ist.psu.edu/sucrow97formal.html`.

B. Sufrin. Formal specification of a display editor. *Science of Computer Programming*, 1:157–202, 1982.

Kenneth J. Turner. *Using Formal Description Techniques — An Introduction to Estelle, LOTOS and SDL*. John Wiley and Sons Ltd., 1993.

Peter H. J. van Eijk, Chris A. Vissers, and Michel Diaz, editors. *The Formal Description Technique LOTOS: Results of the Esprit SEDOS Project*. Elsevier Science, Amsterdam, Netherlands, 1989.

Jos Warmer and Anneke Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 12(1):10–13,28, March 1999.

Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley Professional, 1998.

# Lebenslauf

| | |
|---|---|
| Geboren am: | 24. November 1972 |
| Geburtsort: | Dortmund |
| Eltern: | Peter Beuster<br>Margret Bauer |
| **Abschluss:** | **Abitur** |
| Jahr: | 1992 |
| Institution: | Geschwister-Scholl-Gesamtschule |
| Ort: | Dortmund |
| Note: | 1,4 |
| **Abschluss:** | **Diplom-Informatiker** |
| Jahr: | 2001 |
| Institution: | Universität Koblenz-Landau |
| Ort: | Koblenz |
| Note: | 1,1 |
| **Tätigkeit:** | **Wissenschaftlicher Mitarbeiter** |
| Zeitraum: | April 2001 – August 2002 und April 2003 – April 2007 |
| Arbeitgeber: | Universität Koblenz-Landau |
| Ort: | Koblenz |
| **Stipendium:** | **Doktoranden-Stipendium** |
| Zeitraum: | Oktober 2002 – März 2003 |
| Förderer: | DAAD |
| Ort: | Tschechische Akademie der Wissenschaften<br>Prag, Tschechische Republik |
| **Tätigkeit:** | **Consultant** |
| Zeitraum: | Seit Dezember 2007 |
| Arbeitgeber: | T-Systems Enterprise Services GmbH |
| Ort: | Bonn |